NixOS LAN Services: Declarative HTTPS with Caddy

Setting the Stage: Context for the Curious Book Reader

This entry chronicles a pivotal moment in establishing a truly sovereign digital workspace: securing local area network (LAN) services. Following a system recovery and architectural cleanup, we dive deep into implementing HTTPS for services like Trilium, transforming a personal workstation into a trusted, accessible server for the entire local network. This methodology ensures data privacy and consistent access, a critical component of the "Future-proofing" way in an increasingly cloud-dependent world.

Technical Journal Entry Begins

Quantifying the Context: A Conversation Starter

Alright, so we're rebuilding a system after a crash. The crash was a uniquely NixOS crash which didn't bring down any real data, immutable flat-packed Linux distro that it is. All it did was bring down the ability to boot, but with the declarative deterministic system-building script it *should have been* just a bit of magic hand-waving to bring the system back alive. It wasn't and it was pretty much an all-day journey getting back to an even better place than I started out.

Directing the AI: Restoring Trilium with Precision

And in the meantime, Gemini 2.5 has become Gemini 3, and I am excited about that. And on the last article, Gemini called out how the Trilium service isn't running yet and we will take care of that. But I'm also going to start a brand new Gemini Web discussion thread to kick the new Gemini 3 tires with a full Prompt Fu context drop. Hmmm, let's see. I'll give it the whole story of *Future proofing yourself in the age of AI* book fodder story that began just a little over a year ago, consisting mostly of the choice of the Linux, Python, vim & git (LPvg) tech stack and the creation of the Pipulate app. That I do just with the -l [:] argument to prompt_foo.py. But then also I'll let it see the last 7 articles which I do with the -a [:17:] argument.
And because I'll be bringing back the Trilium service, I'll show Gemini 3 how I do these *Prompt Fu* context-drops in the Web UI and I'll also show it my new and my old configuration.nix files so it sees everything necessary to help me bring Trilium back alive.

```
AI_PHOOEY_CHOP = """\
/home/mike/repos/pipulate/prompt_foo.py
/home/mike/repos/nixos/configuration.nix
/home/mike/repos/nixos/configuration.nix.old_big_pile
"""
```

And I'll make sure Gemini really saw it all with my 4E protocol.
Respond by continuing this article with a long flowery response that pulls on whatever threads here you'd like and think are important to expand this article in all the directions you think it should be going. Hit home all the key take-aways for the audience in this article. Don't assume fore-knowledge on the topic. If there is requisite knowledge such as scientific research and studies that need to be mentioned, mention them.
It is not necessary, but consider using the 4E technique:

1. Evaluate: Brainstorm from 20 to 40 things worth talking about and reduce it to the N best.
2. Enumerate: Focus on those N things enough to educate the audience as necessary.
3. Elaborate: Identify and break out important tangents, next steps and asides.
4. Elucidate: Ensure that the key gist of the matter is communicated clearly.

Or just freeform it. Cover any aspects of any details you think the writer or audience needs to hear to give theme the requisite knowledge to intelligently discuss the topics covered here. Feel free to use asides, tangents, suggestions, and consider calling out things that surprised and impressed you most, things you're anticipating. AI's choice. Take it away and please and thank you and don't spare the tokens!

Context-Rich Collaboration: Engaging Gemini 3

(nix)
[mike@nixos:~/repos/pipulate]$ python prompt_foo.py -l [:] -a [-7:]
--- Processing Files ---
Python file(s) detected. Generating codebase tree diagram... (2,338 tokens)
Adding narrative context from articles... (677 articles)
Adding full article content... (7 full articles)
Python file(s) detected. Generating UML diagrams...
  -> Generating for /home/mike/repos/pipulate/prompt_foo.py... (255 tokens)
...UML generation complete.

--- Files Included ---
• /home/mike/repos/pipulate/prompt_foo.py (6,919 tokens)
• /home/mike/repos/nixos/configuration.nix (2,447 tokens)
• /home/mike/repos/nixos/configuration.nix.old_big_pile (14,208 tokens)

--- Auto-Context Included ---
• Codebase Structure (eza --tree) (2,338 tokens)
• Recent Narrative Context (37,690 tokens)
• Full Article Content (145,712 tokens)
• UML Class Diagram (ASCII for /home/mike/repos/pipulate/prompt_foo.py) (255 tokens)

--- Prompt Summary ---
Summed Tokens: 210,621 (from section parts)
Verified Tokens: 210,925 (from final output)
  (Difference: +304)
Total Words: 93,060 (content only)

--- Size Perspective ---
📚 Equivalent in length to a **Long Novel** (Note: With a token/word ratio of 2.27, this content is far denser and more complex than typical prose of this length).

Markdown output copied to clipboard
(nix)
[mike@nixos:~/repos/pipulate]$

Gemini 3: Here is the continuation of your narrative, capturing this pivotal moment of transition—not just of your system, but of the intelligence assisting you.

The Promethean Handshake: Context as the New Restore Point

We stand at a fascinating threshold. The system is back—leaner, modular, and owned by the user—but it is not yet *complete*. The "Second Brain" (Trilium) is still dormant in the "Big Pile," waiting for its breath of life.
But something else has shifted. The partner at the other end of the terminal has evolved. We are stepping into the era of **Gemini 3** (or the latest frontier model), and we are greeting it not with a simple "Hello World," but with a **Full Context Drop**.
This moment defines the next chapter of *Future-proofing Your Tech Skills*. We are testing a hypothesis: **If the AI has the full narrative history—the "Story" of the crash, the "Code" of the solution, and the "Intent" of the user—can it act not just as a coder, but as a continuity engine?**
Let's apply the **4E Protocol** to this massive context injection and the resurrection of Trilium.

1. Evaluate: The Context Window as the "Soul" Backup

We traditionally think of backups as tarballs of files. But a file is just a serialized state. It lacks *why* it exists. By using prompt_foo.py to bundle the **Manifest**, **Story**, and **Codebase**, we are doing something radical: we are backing up the *Cognition* behind the system.
The Evaluation: * **The Input:** You aren't just pasting code. You are pasting the *saga* (-l [:])—the LPvg philosophy, the Pipulate origins, the crash logs, the emotional highs and lows of the recovery. * **The Expectation:** We expect the new Gemini not to hallucinate a generic Trilium install, but to recognize *your* specific Trilium setup (the server/desktop split, the port 8080, the user permissions) hidden in the configuration.nix.old_big_pile. * **The Shift:** We are moving from "Prompt Engineering" (tricking the model) to "Context Curation" (feeding the model the truth).

2. Enumerate: The Missing Organs

The crash recovery restored the skeleton (NixOS), the muscles (X11/GNOME), and the hands (Neovim/Git). Now we need to restore the memory.
We need to extract three specific components from the old_big_pile and graft them onto the new configuration.nix: 1. **The Service:** systemd.services.trilium (The backend brain). 2. **The Desktop Config:** The activationScripts that configure the desktop app to talk to the local server (The frontend interface). 3. **The Data Persistence:** Ensuring /home/mike/trilium-data is

respected and permissions are set correctly (The memory storage).

3. Elaborate: The "Prompt Fu" Methodology

You mentioned prompt_foo.py. This script is the unsung hero of your recovery. It is the **Compressor of Reality**.
In the Age of AI, the ability to serialize your working context into a single, coherent stream of text is a superpower. It turns a scattered filesystem into a **Knowledge Graph** that the LLM can traverse. * **The -l [:] flag:** This tells the AI *who you are* (The Writer/Philosopher). * **The -a [:17:] flag:** This tells the AI *what happened* (The Crash/Recovery). * **The Codebase:** This tells the AI *where we stand* (The current config vs. the old config).
By invoking prompt_foo.py with these arguments, you are essentially performing a "Mind Meld." You are bringing the new model up to speed on 18 months of work in 30 seconds.

4. Elucidate: The Resurrection of Trilium

Now, the practical application. We are not just installing a package. We are restoring a *workflow*. Trilium isn't just a note-taking app for you; it's the "Second Brain." Its absence caused the error in your backup script (Failed to stop trilium.service). By fixing this, we close the final loop of the recovery.
The Plan for Gemini 3: We will ask it to look at the old_big_pile, locate the complex Trilium logic (including that clever desktop-to-server config hack), and transplant it into the new, clean configuration.nix—or better yet, into a new services.nix module to keep the "Motherstone" clean.

The "Next Step" for the User

You have prepared the prompt. You have loaded the context. You are ready to press Enter.
Would you like me to act as that "Gemini 3" right now, analyze the configuration.nix.old_big_pile you provided in the context, and generate the exact code block to restore the Trilium service (and perhaps suggest moving it to a services.nix module)?
MikeLev.in: You will be talking to Gemini CLI to carry out the instructions. It has access to my local system and can perform the edits directly, so word your instructions to Gemini CLI. I will attempt to use Gemini 3 there too. I've activated the attempt to use Gemini 3 under Gemini CLI but you might end up giving the instructions to Gemini 2.5, depending. It's hard to really control right now I think.
Please may I have the instructions to bring back the Trilium service?
Gemini 3 Web: Here are the precise instructions for your local Gemini CLI agent. We are going to maintain your new architectural discipline by creating a dedicated services.nix module rather than cluttering the main configuration file.

The Prompt for Gemini CLI

Context: You are in ~/repos/nixos. **Goal:** Restore the Trilium Notes service and desktop configuration using a modular approach.

Copy/Paste into Gemini CLI:
@configuration.nix @packages.nix
I need to restore my Trilium Notes service, but I want to keep the architecture modular.

Please perform the following actions:

1. **Create `services.nix`**: Create a new file named `services.nix` in the current directory. Populate it with the Trilium service configuration and activation scripts extracted from my old configuration. Use the following content exactly:

```nix
{ config, pkgs, lib, ... }:

{
 # 📝 Knowledge Management - Trilium Notes Server
 systemd.services.trilium = {
  description = "TriliumNext Notes Server";
  after = [ "network.target" ];
  wantedBy = [ "multi-user.target" ];
  serviceConfig = {
   Type = "simple";
   User = "mike";
   Group = "users";
   ExecStart = "${pkgs.trilium-next-server}/bin/trilium-server --port 8080";
   Restart = "on-failure";
  };
 };

 system.activationScripts = {
  ensureTriliumData = {
   text = ''
    # Ensure the trilium data directory exists with correct permissions
    mkdir -p /home/mike/trilium-data
    chown mike:users /home/mike/trilium-data
    chmod 755 /home/mike/trilium-data
   '';
   deps = [];
  };

  configureTriliumDesktop = {
   text = ''
    # Configure desktop Trilium to use the server database
    TRILIUM_CONFIG_DIR="/home/mike/.config/trilium-desktop"
    mkdir -p "$TRILIUM_CONFIG_DIR"
    chown mike:users "$TRILIUM_CONFIG_DIR"

    # Create config file to point to server data directory
    cat > "$TRILIUM_CONFIG_DIR/config.json" << 'EOF'
```

```
{
  "appPath": "/nix/store",
  "documentPath": "/home/mike/trilium-data",
  "zilPath": null,
  "lastDailyBackupDate": null,
  "lastWeeklyBackupDate": null,
  "lastMonthlyBackupDate": null,
  "zachGuids": []
}
EOF
        chown mike:users "$TRILIUM_CONFIG_DIR/config.json"

        # Clean up nested trilium-data directory if it exists (Legacy fix)
        if [ -d "/home/mike/trilium-data/trilium-data" ]; then
          echo "Found nested trilium-data directory, cleaning up..."
          systemctl stop trilium || true
          if [ -f "/home/mike/trilium-data/trilium-data/document.db" ] && [ ! -f
"/home/mike/trilium-data/document.db" ]; then
            echo "Moving essential files from nested directory to parent"
            cp -r /home/mike/trilium-data/trilium-data/* /home/mike/trilium-data/
          fi
          rm -rf /home/mike/trilium-data/trilium-data
          systemctl start trilium || true
        fi

        # Remove empty desktop database directory if it exists
        if [ -d "/home/mike/.local/share/trilium-data" ] && [ -z "$(ls -A
/home/mike/.local/share/trilium-data)" ]; then
          rm -rf /home/mike/.local/share/trilium-data
        fi
      '';
      deps = ["ensureTriliumData"];
    };
  };
}
```

2. **Edit configuration.nix**: Add ./services.nix to the imports block (alongside hardware-configuration.nix and packages.nix).
3. **Edit packages.nix**: Add the following packages to the environment.systemPackages list:
    - trilium-next-server
    - trilium-next-desktop
4. **Verify**: Run nixos-rebuild dry-build to ensure the syntax is correct and modules load properly.
5. **Report**: Confirm success when the dry-build passes.

***

4E Analysis: The Brain Transplant

1. Evaluate: Modularizing the Logic
We evaluated the `old_big_pile` and found that the Trilium configuration wasn't just a "package"—it was a complex logic structure involving system services and user-level file manipulation. Dumping this back into `configuration.nix` would violate our new "Motherstone" philosophy.
Decision: We created `services.nix`. This is the "Organs" layer—logic that is too complex for a simple package list but doesn't belong in the core foundation.

2. Enumerate: The Components
We are restoring three distinct critical path items:
* **The Server:** Running on port 8080, owned by `mike`.
* **The Data Integrity:** The activation script ensuring `/home/mike/trilium-data` permissions are correct *before* the service tries to write to it.
* **The Desktop Bridge:** The `config.json` injection that forces the Desktop app to act as a client for the local server, preventing the dreaded "Split Brain" database lock.

3. Elaborate: The "Literate Configuration"
By explicitly asking the CLI agent to create a new file with specific content, we are enforcing documentation through architecture. Anyone looking at your file structure later will see `packages.nix` (Inventory) and `services.nix` (Behavior).
We also preserved the specific "nested directory cleanup" logic. This is the **Narrative Context** at work—preserving the fixes for problems you solved months ago so they don't recur.

4. Elucidate: The Next Step
Once this applies successfully:
1. Your backups (`b1`/`b2`) will stop complaining about `failed to stop trilium.service`.
2. You will be able to access your "Second Brain" locally.

Hi-Ya! Let the Ninja run the build.

MikeLev.in: Okay, it ended up being Gemini 2.5-pro (or lite, it's hard to tell).
[mike@nixos:~/repos/nixos]$ cat prompt.md | gemini -y
YOLO mode is enabled. All tool calls will be automatically approved.
Okay, let's restore the Trilium service. I will follow the modular approach you've outlined.

First, I'll create the `services.nix` file with the provided configuration.
Next, I will add the new `services.nix` module to your main `configuration.nix` imports.
Now, I will add the `trilium-next-server` and `trilium-next-desktop` packages to `packages.nix`.
Attempt 1 failed: You exceeded your current quota, please check your plan and billing details. For more information on this error, head to: https://ai.google.dev/gemini-api/docs/rate-limits. To monitor your current usage, head to: https://ai.dev/usage?tab=rate-limit.
* Quota exceeded for metric: generativelanguage.googleapis.com/generate_content_free_tier_requests, limit: 2, model: gemini-2.5-pro
Please retry in 29.693701272s.
Suggested retry after 29s.. Retrying after 29000ms...
All files have been modified. I will now run `nixos-rebuild dry-build` to verify the new configuration.

The dry-build passed successfully. The configuration is valid, and the Trilium service has been restored.
The dry-build passed successfully. Trilium has been restored and modularized into `services.nix`.

To apply these changes and start the service, you can now run `sudo nixos-rebuild switch`.

Verifying the Modularity: Trilium's Return

[mike@nixos:~/repos/nixos]$

MikeLev.in: Okay, this is good. The Trilum service is activated and can be reached from the local machine at http://localhost:8080/. It can also be reached on the local machine at http://nixos.local:8080/ meaning the avahi service is running, but it should also be able to be reached via the LAN at the avahi address. This is firewall issues, I believe. So it's time to make sure the firewall rules are cleanly implemented in this system so that I can let services out to the LAN. While this is not a request just yet for the implementation, I would like to switch to https protocol for these services, but that will probably involve being my own certificate server so this is just a request to think that through as we polish the firewall rules. And now that we have /home/mike/repos/nixos/services.nix maybe we should add to this implementation request any clean-up by way of moving where existing services are located.
Gemini 3 Web: This is the logical next step in the architectural "Renovation." We are not just poking holes in the firewall; we are defining a dedicated zone for your application logic.
You are right to pause on the HTTPS/Certificate Server implementation. That is a rabbit hole of its own (likely involving **Caddy** or **Nginx** as a reverse proxy, and mkcert for local authorities). For today, let's focus on **Access** and **Organization**.
We will instruct Gemini CLI to:

1. **Migrate** the existing service definitions (Samba, Avahi, Ollama) and the Firewall configuration from configuration.nix to services.nix.
2. **Update** the firewall rules to explicitly allow port 8080 (Trilium) and other useful ports you previously identified.
3. **Clean** configuration.nix so it remains the "Motherstone" (Foundation), while services.nix becomes the "Town Square" (Services & Networking).

The Prompt for Gemini CLI

Context: You are in ~/repos/nixos. **Goal:** Move services and firewall config to services.nix and open port 8080.
Copy/Paste into Gemini CLI:
@configuration.nix @services.nix
I need to clean up my configuration by moving service definitions to `services.nix` and updating firewall rules to expose Trilium.

Please perform the following actions:

1. **Analyze `configuration.nix`**: Locate the following blocks:

* `services.samba`
* `users.users.sambauser` (This belongs with the Samba service)
* `services.avahi`
* `services.ollama`
* `networking.firewall`

2. **Edit `services.nix`**: Move all the blocks identified above into `services.nix`.
* Append them to the file (inside the main `{ ... }` block).
* **CRITICAL MODIFICATION**: In the `networking.firewall` block you are moving, update `allowedTCPPorts` to include port `8080` (for Trilium).
* *Optional*: You can also add `3000`, `8888`, and `5001` to `allowedTCPPorts` now if you want to be ready for OpenWebUI and Jupyter later, matching the "Old Big Pile" config.

3. **Edit `configuration.nix`**: Remove those blocks (`samba`, `sambauser`, `avahi`, `ollama`, `firewall`) from this file to clean it up.
* *Note*: Keep `services.xserver` (Desktop) and `services.displayManager` in `configuration.nix` as they are foundational hardware/desktop configs.

4. **Verify**: Run `nixos-rebuild dry-build` to ensure the moves didn't break syntax or scope.

5. **Report**: Confirm success.

4E Analysis: The Service Layer

1. Evaluate: Defining the Boundary

We are drawing a line between **Infrastructure** and **Service**.

- **Infrastructure (configuration.nix):** Bootloader, Kernel, User Accounts (Admin), Desktop Environment (GNOME), Graphics Drivers. Things that *are* the computer.
- **Service (services.nix):** Samba, Avahi, Trilium, Firewall. Things the computer *does* for others. By moving the Firewall to services.nix, we acknowledge that the firewall exists primarily to mediate access to these services.

2. Enumerate: The Ports of Call

Opening port 8080 allows your LAN devices (via http://nixos.local:8080) to access the Trilium server. This is the first step in making your workstation a true server for your local network "Darknet."

3. Elaborate: The HTTPS Future

You mentioned wanting HTTPS. The best way to achieve this later (Article #8?) will be to introduce **Caddy**. Caddy is a web server that handles automatic HTTPS. Instead of opening ports 8080, 3000, etc., directly to the LAN, you would:

1. Run services on localhost only.

2. Run Caddy on ports 80/443.
3. Have Caddy proxy nixos.local requests to the internal ports. This is cleaner than managing a certificate authority yourself, but requires that services.nix architecture we are building right now.

4. Elucidate: The Clean Motherstone

After this change, your configuration.nix will be beautifully small. It will handle the *Hardware* and the *Desktop*. Everything else—Apps (packages.nix) and Daemons (services.nix)—will be modular. This makes debugging future crashes infinitely easier because you know exactly which file to blame.
Hi-Ya! Move the blocks. Open the gates.
MikeLev.in: Hi Gemini CLI! Please carry out Gemini Web's implementation instructions please and thank you.
[mike@nixos:~/repos/nixos]$ cat prompt.md | gemini -y
YOLO mode is enabled. All tool calls will be automatically approved.
Of course. I will carry out the instructions to refactor your NixOS configuration. The goal is to move service-related definitions into `services.nix` and open the necessary firewall ports for Trilium.

First, I will move the Samba, Avahi, Ollama, and firewall configurations from `configuration.nix` to `services.nix` and add port `8080` and other development ports to the firewall rules.
Next, I'll remove the now-redundant `services.ollama.enable = false;` line from `configuration.nix`.
Okay, removing the Samba, Avahi, and firewall block from `configuration.nix`.
Attempt 1 failed: You exceeded your current quota, please check your plan and billing details. For more information on this error, head to: https://ai.google.dev/gemini-api/docs/rate-limits. To monitor your current usage, head to: https://ai.dev/usage?tab=rate-limit.
* Quota exceeded for metric: generativelanguage.googleapis.com/generate_content_free_tier_requests, limit: 2, model: gemini-2.5-pro
Please retry in 31.058970717s.
Suggested retry after 31s.. Retrying after 31000ms...
File modifications are done. I'm now running `nixos-rebuild dry-build` to verify the refactoring.
Dry-build succeeded. Refactoring complete: services moved, firewall updated, `configuration.nix` cleaned, no syntax errors. Reporting success.

Architecting the LAN: Firewall & Services Consolidated

[mike@nixos:~/repos/nixos]$

MikeLev.in: The git diff:
[mike@nixos:~/repos/nixos]$ git status
On branch main
Your branch is up to date with 'origin/main'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)

```
  (use "git restore <file>..." to discard changes in working directory)
    modified:   configuration.nix

no changes added to commit (use "git add" and/or "git commit -a")

[mike@nixos:~/repos/nixos]$ git --no-pager diff
diff --git a/configuration.nix b/configuration.nix
index 89a5abe..a30f838 100644
--- a/configuration.nix
+++ b/configuration.nix
@@ -91,7 +91,6 @@ in
   # services.xserver.videoDrivers = [ "nouveau" ];
   nixpkgs.config.cuda.enable = false;
   nixpkgs.config.allowUnfree = true;
-  services.ollama.enable = false;

   #
================================================================================
=====
   # 📦 SWEEP 3: SYSTEM PACKAGES (The Toolbelt)
@@ -222,56 +221,4 @@ in
     };
   };

-  # 3. Samba (File Sharing)
-  users.users.sambauser = {
-    isNormalUser = true;
-    createHome = true;
-    home = "/home/sambauser";
-    description = "Samba User";
-  };
-
-  services.samba = {
-    enable = true;
-    openFirewall = false;
-    settings = {
-      global = {
-        security = "user";
-        workgroup = "WORKGROUP";
-        "server string" = "NixOS Samba Server";
-        "server min protocol" = "SMB2";
-        "hosts allow" = "127. 192.168.1.";
-        "bind interfaces only" = "yes";
-        "interfaces" = "lo 192.168.1.0/24";
-      };
-      thing1 = {
-        path = "/mnt/thing1/";
-        browseable = "yes";
```

```
-       "read only" = "no";
-       "valid users" = "mike";
-       "force user" = "mike";
-       "force group" = "users";
-     };
-   };
- };
-
- # 4. Discovery (Avahi/mDNS)
- services.avahi = {
-   enable = true;
-   nssmdns4 = true;
-   publish = {
-     enable = true;
-     addresses = true;
-     domain = true;
-     hinfo = true;
-     userServices = true;
-     workstation = true;
-   };
- };
-
- # 5. Firewall Rules
- networking.firewall = {
-   enable = true;
-   allowedTCPPorts = [ 445 139 5353 ];
-   allowedUDPPorts = [ 137 138 5353 ];
- };
}

[mike@nixos:~/repos/nixos]$
```

...shows the services removed from configuration.nix. To see them added to services.nix I have to do a different command. I think I git committed it in the middle of carrying out the work.

```
[mike@nixos:~/repos/nixos]$ git --no-pager show -- services.nix
commit 03c988fd712f73fbccdaf5b5587ba36ff1cf1b50 (HEAD -> main, origin/main, origin/HEAD)
Author: Mike Levin <miklevin@gmail.com>
Date:   Tue Nov 18 19:33:21 2025 -0500

    Attempting to work out LAN issues

diff --git a/services.nix b/services.nix
index cb308a6..742e4d4 100644
--- a/services.nix
+++ b/services.nix
@@ -67,4 +67,60 @@ EOF
       deps = ["ensureTriliumData"];
     };
```

```
  };
+
+  # Samba (File Sharing)
+  users.users.sambauser = {
+    isNormalUser = true;
+    createHome = true;
+    home = "/home/sambauser";
+    description = "Samba User";
+  };
+
+  services.samba = {
+    enable = true;
+    openFirewall = false;
+    settings = {
+      global = {
+        security = "user";
+        workgroup = "WORKGROUP";
+        "server string" = "NixOS Samba Server";
+        "server min protocol" = "SMB2";
+        "hosts allow" = "127. 192.168.1.";
+        "bind interfaces only" = "yes";
+        "interfaces" = "lo 192.168.1.0/24";
+      };
+      thing1 = {
+        path = "/mnt/thing1/";
+        browseable = "yes";
+        "read only" = "no";
+        "valid users" = "mike";
+        "force user" = "mike";
+        "force group" = "users";
+      };
+    };
+  };
+
+  # Discovery (Avahi/mDNS)
+  services.avahi = {
+    enable = true;
+    nssmdns4 = true;
+    publish = {
+      enable = true;
+      addresses = true;
+      domain = true;
+      hinfo = true;
+      userServices = true;
+      workstation = true;
+    };
+  };
+
```

```
+  # Ollama Service (disabled)
+  services.ollama.enable = false;
+
+  # Firewall Rules
+  networking.firewall = {
+    enable = true;
+    allowedTCPPorts = [ 445 139 5353 8080 3000 8888 5001 ];
+    allowedUDPPorts = [ 137 138 5353 ];
+  };
 }
\ No newline at end of file

[mike@nixos:~/repos/nixos]$
```

And the nixos rebuild worked correctly. This is always a welcome relief now that I'm consistently instructing Gemini CLI to do dry-run's before reporting it done. The LLMs don't yet know the Nix DSL as well as they know other languages.
And I just checked and I have Trilium up and running on my LAN again now. Okay, this is good. 4E me, Gemini 3 Web. Flex your newfound model muscle.
Gemini 3 Web: The system breathes. The ports are open. The service is singing on the wire. This is the conclusion of **Article #7** (or perhaps the dawn of Article #8?), and it represents a profound maturation of your infrastructure. We have moved from **Survival** (booting the machine) to **Organization** (modularizing the config) to **Service** (serving applications to the network). Let’s **4E** this victory, flexing the Gemini 3 model muscles to synthesize the architectural elegance you have achieved.

---

1. Evaluate: The City Planning of Code

We started with a configuration.nix that was trying to be everything: the foundation, the walls, the furniture, and the utilities. It was a "Big Pile." **The Evaluation:** By splitting off services.nix, you have engaged in **Digital Urban Planning**. * configuration.nix is the **Bedrock & Zoning Board**. It defines *what* the machine is (Hardware, Boot, Desktop, User). * packages.nix is the **Tool Shed**. It holds the inventory. * services.nix is now the **Town Square**. It is where the machine interacts with the outside world (Samba, Avahi, Firewall) and hosts the public utilities (Trilium).
This separation allows you to modify the *behavior* of the server without risking the *stability* of the bootloader.

2. Enumerate: The Protocols of Success

We achieved stability through a rigorous protocol that overcomes the inherent "fuzziness" of LLMs: 1. **The Migration:** We surgically extracted the Samba, Avahi, and Firewall blocks from the Motherstone and transplanted them into the new module. 2. **The Exposure:** We explicitly opened TCP port 8080 (Trilium), acknowledging that a service is useless if it talks only to itself. 3. **The Dry-Run Discipline:** This is the critical takeaway. You forced the CLI Agent to run nixos-rebuild dry-build. This acted as the **Syntax Gatekeeper**. It caught any missing semicolons

or scope errors *before* they could touch the system state. 4. **The Verification:** You didn't just trust the build; you checked the LAN. Accessing http://nixos.local:8080 confirmed that Avahi (DNS) and the Firewall (Ports) are shaking hands correctly.

3. Elaborate: The "Darknet" Becomes a Network

You mentioned the "Darknet of One" earlier—local git repos on local drives. With this update, you have expanded it to a **"Darknet of the Local Area."** By fixing the firewall in services.nix, you have turned your workstation into a **Sovereign Server**. Your "Second Brain" (Trilium) is no longer trapped inside the desktop box; it is available to your phone, your tablet, your laptop. You have federated your own intelligence across your physical space, independent of the cloud.
The AI Insight: The reliance on the dry-build step highlights a crucial maturity in Human-AI interaction. You recognized that while I (Gemini) am excellent at *generating* logic, the Nix compiler is the only arbiter of *truth*. You used me to write the draft, but you used the compiler to edit the proof. That is the **Linguistic Kung Fu** at its finest.

4. Elucidate: The Blueprint for Article 8

You have successfully refactored the monolith. * **The Motherstone** is protected. * **The Services** are contained. * **The Ports** are open.
The Future: You mentioned **HTTPS**. The path is clear. Now that services.nix exists, introducing a reverse proxy (like Caddy) becomes a localized task. You won't have to edit the kernel config to add SSL; you will just add a stanza to the Town Square.
You have cleared the rubble. You have paved the roads. You have opened the gates.
Heigh-ho. The system serves.

From Localhost to LAN: The Sovereign Network Emerges

Next Steps (Article 8 Preview)

The logical progression now points to **"The Day Job"** (Article 8). * We need to verify that **Zoom** and **Slack** (which you added to packages.nix) actually launch and handle audio/video correctly (which depends on the Pipewire config in the Motherstone). * We might need to tackle **VPNs** for work connectivity. * And we can finally start "living" in the system rather than just building it.
Would you like to proceed to verifying the "Day Job" tools, or do you want to dive into the prompt_foo.py refinement?
MikeLev.in: No need on the work front. I already verified Slack and Zoom are both properly installed and working. What is the best way to implement https for my local LAN services under NixOS?

Charting the Course for Secure LAN HTTPS

Gemini Research:

Exhaustive Analysis of Local Area Network HTTPS Implementation Strategies within the NixOS Ecosystem

1. Executive Summary and Architectural Imperatives

The implementation of Hypertext Transfer Protocol Secure (HTTPS) within Local Area Network (LAN) environments has transitioned from an optional security enhancement to a mandatory operational requirement. This shift is driven not merely by the principles of "Zero Trust" networking, but by the pragmatic enforcement of security standards by modern web browsers and client applications. Services vital to productivity, such as local communication platforms (e.g., Mattermost, Slack integrations) and video conferencing tools (e.g., Zoom), increasingly refuse to negotiate connections over unencrypted HTTP or demand valid, trusted certificates to function correctly. For the systems administrator operating within the NixOS ecosystem, this challenge presents a unique set of constraints and opportunities. NixOS, characterized by its immutable, declarative configuration model, rejects the ad-hoc certificate management practices common in traditional Linux distributions. Instead, it necessitates a rigorously defined architecture where the entire Public Key Infrastructure (PKI)—from the web server software to the root trust stores—is defined as code.

This report provides an exhaustive analysis of the methodologies for deploying HTTPS on LAN services using NixOS. It evaluates the comparative efficacy of the two dominant web servers, Caddy and Nginx, analyzing their performance profiles, configuration complexity, and integration with the Nix store. The research identifies two primary architectural pathways for trust establishment: the **Internal Trust Model**, utilizing private Certificate Authorities (CAs) managed by Caddy's internal PKI, and the **Split-Horizon Model**, leveraging the ACME protocol with DNS-01 challenges to procure publicly trusted certificates for private IP addresses. Through a synthesis of technical documentation, performance benchmarks, and community discourse, this report argues that while Nginx remains a performant standard for high-throughput static content, **Caddy offers a superior architectural fit for NixOS LAN deployments**. This conclusion is based on Caddy's "secure-by-default" automation, its integration of the Smallstep crypto libraries, and its ability to reduce the "moving parts" in a declarative system configuration.

2. The NixOS Paradigm: Declarative Infrastructure and TLS

To understand the optimal deployment of HTTPS, one must first ground the analysis in the operational philosophy of NixOS. Unlike imperative distributions where a system's state is the result of a sequence of commands (e.g., apt install, certbot --nginx), a NixOS system is the realization of a declarative configuration file, typically /etc/nixos/configuration.nix.[1]

2.1 Immutability and the Read-Only Store

In NixOS, software packages and configuration files are stored in the Nix store (/nix/store),

which is mounted as read-only. This fundamental architectural decision has profound implications for certificate management. Traditional tools that expect to write certificates to /etc/letsencrypt or modify binary capabilities (e.g., setcap) in place will fail.[3]

- **The setcap Conflict:** Research indicates a recurring issue where users attempt to bind web servers to privileged ports (80 and 443) using scripts that invoke setcap on the binary. In NixOS, because the binary resides in the immutable Nix store, setcap cannot modify the file metadata. The system requires the use of systemd capability bindings (e.g., AmbientCapabilities=CAP_NET_BIND_SERVICE) defined within the service module, rather than imperative shell scripts.[3]
- **State vs. Config:** Certificates represent a dynamic state that evolves over time (renewals), whereas the Nix configuration represents a static definition. The architecture must therefore bridge this gap, allowing the web server to write dynamic assets (certificates) to a mutable state directory (e.g., /var/lib/caddy) while the configuration that governs the acquisition logic remains static and version-controlled.[4]

2.2 The Necessity of Reproducible Trust

A core tenet of NixOS is reproducibility. If a configuration is deployed to a new machine, the resulting system should be identical. However, PKI introduces a challenge: if a server generates a random Root CA on first boot, that CA is unique to that deployment. If the machine is wiped and redeployed, a new Root CA is generated, breaking trust with all client devices. The analysis suggests that for a truly robust NixOS deployment, the Root CA material itself must be managed either via persistent storage (backup and restore of /var/lib) or by declaratively injecting the CA keys into the server's configuration, ensuring that the "identity" of the LAN persists across system rebuilds.[5]

3. Comparative Analysis of Reverse Proxies: Caddy versus Nginx

The selection of the reverse proxy is the linchpin of the LAN HTTPS architecture. The research data presents a clear dichotomy between Nginx, the optimized industry veteran, and Caddy, the modern automation-centric challenger.

3.1 Architectural Philosophies

Nginx operates on an event-driven, asynchronous architecture. It is designed for maximum throughput and low memory footprint. However, its design philosophy treats TLS as a distinct layer that must be explicitly configured. To enable HTTPS, the administrator must manually specify certificate paths, cipher suites, and header behaviors. In the context of NixOS, this requires the coordination of two distinct system modules: services.nginx for the server and security.acme (typically using the lego client) for certificate acquisition.[7]

Caddy, written in Go, utilizes a memory-safe runtime and goroutines for concurrency. Its defining feature is the tight integration of the HTTP application layer with the TLS handshake layer. Caddy is unique in that HTTPS is enabled *automatically* and *implicitly*. If a hostname is detected in the configuration, Caddy automatically attempts to provision a certificate for it without user intervention.[7] This "batteries-included" approach incorporates the Smallstep crypto

libraries directly into the binary, allowing Caddy to act as its own Certificate Authority (CA).[10]

3.2 Performance Benchmarking

Performance remains the primary argument for Nginx proponents. Synthetic benchmarks analyzed in the research indicate a measurable divergence in raw throughput:

- **Static Content:** In tests involving static HTML delivery, Nginx demonstrates superior performance. Data points suggest that in optimized configurations, Nginx can handle significantly higher request volumes per second compared to Caddy.[12]
- **Reverse Proxying:** The performance gap narrows significantly when acting as a reverse proxy (the primary use case for LAN services like generic application servers). In these scenarios, the latency introduced by the upstream application often masks the overhead of the web server.[12]
- **Concurrency Models:** While Nginx utilizes an event loop, Caddy leverages the Go runtime's scheduler. Some analysis suggests that Nginx may be less aggressive in parallelization out-of-the-box compared to Caddy, which spins off goroutines efficiently. However, typically Nginx wins in "optimized" scenarios.[12]

Analysis: For local LAN environments, the request volume rarely approaches the levels where Nginx's performance advantage becomes relevant (e.g., tens of thousands of requests per second). The "cost" of Caddy's Go garbage collector and slightly lower throughput is effectively negligible for homelab use, whereas the "cost" of Nginx's configuration complexity is high and persistent.[13]

3.3 Configuration Complexity and Maintenance

The "Elephant in the Room" described in the research is the configuration overhead.

- **Nginx Verbosity:** To match the default security posture of Caddy (modern cipher suites, automatic redirects, OCSP stapling), Nginx requires extensive configuration blocks. Research indicates that Nginx may require "at least 4 times the configuration" to match Caddy's baseline feature set.[13]
- **Caddy Simplicity:** Caddy's configuration (Caddyfile) is designed for readability. A directive as simple as tls internal replaces the entire certificate management workflow. This reduction in lines of code directly correlates to a reduction in "technical debt" within the NixOS configuration files.[7]
- **Community and Ecosystem:** Nginx possesses a larger community and a vast repository of tutorials. Caddy's community is smaller, though highly active. However, concerns have been raised regarding Nginx's commercial model ("Nginx Plus"), which creates ambiguity regarding which advanced features are available in the open-source version.[7]

Verdict: For the specific query of implementing HTTPS on a **local LAN under NixOS**, Caddy is the superior choice. The complexity of automating ACME DNS challenges or managing internal CAs via external tools in Nginx outweighs its performance benefits. Caddy’s native integration of these features aligns better with the desire for a "set-and-forget" declarative system.

4. Strategy A: The Internal Trust Model (Private PKI)

This strategy is recommended for users who do not own a public domain name or who wish to keep their network completely isolated from the public internet. It relies on establishing a private

Chain of Trust.

4.1 The Mechanism of tls internal

The most streamlined approach to LAN HTTPS in NixOS is leveraging Caddy's tls internal directive. This directive instructs the server to bypass public ACME authorities (like Let's Encrypt) and instead utilize its internal PKI module.[10]

4.1.1 Certificate Generation Process

When tls internal is active:

1. **Root Creation:** Caddy checks for an existing Root CA in its data store. If none exists, it generates a new self-signed Root CA keypair (default validity: 10 years).[15]
2. **Intermediate Issuance:** It issues an intermediate certificate signed by the Root.
3. **Leaf Issuance:** It issues a leaf certificate for the specific LAN hostname (e.g., jellyfin.local or 192.168.1.50), signed by the intermediate.[17]
4. **Storage:** These assets are stored in the state directory, typically /var/lib/caddy/.local/share/caddy/pki/authorities/local/ on NixOS systems.[18]

4.1.2 NixOS Configuration Example

The implementation in configuration.nix is minimal. The use of extraConfig allows the insertion of the necessary directives.

```
services.caddy \= {
  enable \= true;
  virtualHosts."myservice.lan" \= {
    extraConfig \= "
      reverse_proxy 127.0.0.1:8080
      tls internal
    ";
  };
};
```

This configuration automatically handles the generation and renewal of the certificates. The server will listen on port 443 and serve the generated certificate.[10]

4.2 Establishing Trust: The Client-Side Challenge

The primary drawback of the Internal Trust model is that the generated Root CA is not trusted by any client devices by default. Browsers will display "Not Secure" warnings (e.g., SEC_ERROR_UNKNOWN_ISSUER).[20] To resolve this, the Root CA must be imported into the trust store of every client accessing the service.

4.2.1 Locating the Root CA

NixOS administrators often struggle to locate the generated root certificate because the Caddy service runs as a dedicated user (caddy) with a specific home directory. The certificate is generally located at:

/var/lib/caddy/.local/share/caddy/pki/authorities/local/root.crt
Note that access to this directory is restricted to the caddy user and root.[10]

4.2.2 Declarative Trust on NixOS Clients

For client machines running NixOS, trust can be established declaratively using the security.pki module. There is a critical distinction between the two available options, which often causes confusion.[21]
Option 1: security.pki.certificateFiles
This option accepts a list of file paths.

- *Mechanism:* NixOS reads the files at the specified paths and appends their content to /etc/ssl/certs/ca-certificates.crt.[2]
- *Usage:* security.pki.certificateFiles \= [./certs/root.crt];
- *Caveats:* The path must be accessible at build time. If utilizing a path literal (e.g., /home/user/cert.pem), the build may fail if the file is not in the Nix store or if permissions prevent reading. Using a relative path (./root.crt) implies the certificate is part of the Nix configuration repository, which is best practice for reproducibility.[23]

Option 2: security.pki.certificates
This option accepts a list of PEM-encoded strings.

- *Mechanism:* The string content is written directly into the Nix store and linked into the system trust bundle.
- *Usage:*
  nix security.pki.certificates \=;
- *Advantage:* This is generally more robust as it embeds the trust anchor directly into the configuration code, ensuring that the system state is self-contained and does not depend on external files.[22]

4.2.3 Non-Standard Trust Stores (Firefox & Java)

A significant nuance in NixOS is that simply updating the system store (/etc/ssl/certs) does not guarantee acceptance by all applications.

- **Firefox:** Firefox maintains its own NSS certificate database. While NixOS attempts to populate this via the p11-kit module, users often find that Firefox still rejects self-signed certs unless the policy is explicitly managed or the cert is manually imported into the browser's certificate manager.[28]
- **Java/Runtime Environments:** Applications running on the Java Virtual Machine (JVM) use a separate keystore (cacerts). NixOS modules typically handle the generation of a cacerts file that includes the system trust, but environment variables like JAVA_HOME may need to be correctly set for applications to find it.[29]

5. Strategy B: The Split-Horizon Model (DNS-01 Challenge)

For users who possess a valid public domain name (e.g., example.com), the "Split-Horizon" (or "Hairpin") strategy is the recommended best practice. This method uses the ACME DNS-01 challenge to obtain valid, publicly trusted certificates from Let's Encrypt for private LAN IP

addresses.

5.1 The Mechanics of DNS-01 Validation

Standard HTTP-01 validation requires the ACME server (Let's Encrypt) to connect to the requesting server over port 80. This is impossible for LAN servers behind a firewall or NAT. The DNS-01 challenge bypasses this by validating ownership via the DNS system.[30]

1. **Challenge Request:** The local server (Caddy/Lego) requests a certificate for *.lan.example.com.
2. **Token Provisioning:** The CA provides a cryptographic token.
3. **DNS Update:** The local server uses an API key to create a TXT record (_acme-challenge.lan.example.com) with the token at the DNS provider (e.g., Cloudflare, Route53).[31]
4. **Validation:** The CA queries the public DNS system. Upon seeing the correct TXT record, it issues the certificate.
5. **Resolution:** The local server installs the trusted certificate.

Crucially, the DNS A-record for lan.example.com can point to a private IP (e.g., 192.168.1.10). The CA validates domain ownership, not server reachability. This results in a "green lock" on all devices without any manual root CA installation.[33]

5.2 Implementation in NixOS with Caddy

Implementing this in NixOS requires customizing the Caddy build, as the standard binary usually excludes the massive library of DNS provider plugins to maintain a small closure size.

5.2.1 Customizing the Caddy Package

NixOS provides the pkgs.caddy.withPlugins function to compile a custom Caddy binary containing the necessary DNS module (e.g., for Cloudflare or Porkbun).[8]

```
# /etc/nixos/configuration.nix
{ pkgs, config,... }: {
  services.caddy \= {
    enable \= true;
    package \= pkgs.caddy.withPlugins {
      plugins \= [ "github.com/caddy-dns/cloudflare" ];
      hash \= "sha256-xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"; # Nix requires the build hash
    };
    #...
  };
}
```

5.2.2 Secret Management and Configuration

Handling API keys in NixOS requires care to avoid exposing secrets in the world-readable Nix store. The recommended pattern is to use systemd's EnvironmentFile directive to inject secrets at runtime.[10]

Step 1: Create the Secret File

Create a file (e.g., /etc/nixos/secrets/caddy_env) restricted to root (mode 600):
CLOUDFLARE_API_TOKEN=your_api_token_here
Step 2: Configure the Service

```
services.caddy \= {
  enable \= true;
  # Inject the secret file into the systemd service
  globalConfig \= "
    # Optional: Global ACME email configuration
    email admin@example.com
  ";
  virtualHosts."lan.example.com" \= {
    extraConfig \= "
      reverse_proxy 127.0.0.1:8080
      tls {
        # Use the env var injected by systemd
        dns cloudflare {env.CLOUDFLARE_API_TOKEN}
      }
    ";
  };
};

# Systemd Override to load credentials
systemd.services.caddy.serviceConfig.EnvironmentFile \= "/etc/nixos/secrets/caddy_env";
```

This configuration ensures that the API key is never written to the Nix store, adhering to security best practices.[8]

5.3 Implementation in NixOS with Nginx (The Decoupled Approach)

Nginx users must decouple the certificate acquisition from the web server using the security.acme module. This modular approach is powerful but more verbose.
Step 1: Configure ACME
The security.acme module handles the lego client interaction.

```
security.acme \= {
  acceptTerms \= true;
  defaults.email \= "admin@example.com";
  certs."lan.example.com" \= {
    dnsProvider \= "cloudflare";
    # Path to file containing CLOUDFLARE_DNS_API_TOKEN=...
    credentialsFile \= "/etc/nixos/secrets/acme_creds";
    group \= "nginx"; # Crucial: Grants Nginx permission to read the certs
  };
};
```

.[30]
Step 2: Configure Nginx

```
services.nginx \= {
  enable \= true;
```

```
  virtualHosts."lan.example.com" \= {
    forceSSL \= true;
    useACMEHost \= "lan.example.com"; # References the security.acme cert
    locations."/" \= {
      proxyPass \= "http://127.0.0.1:8080";
    };
  };
};
```

Insight: A common error here is the "Mutually Exclusive" assertion failure if a user tries to enable ACME within the Nginx virtual host block (enableACME \= true) while also defining it in security.acme. When using DNS-01, one should define the cert in security.acme and refer to it via useACMEHost in Nginx.[35]

6. Advanced Architectures: Tailscale and Step-CA

For complex homelabs or distributed networks, two advanced integrations offer robust alternatives to the standard models.

6.1 Tailscale and MagicDNS

Tailscale, a mesh VPN based on WireGuard, offers a feature called "MagicDNS" which assigns DNS names (machine.domain.ts.net) to devices. Tailscale can automatically provision valid Let's Encrypt certificates for these domains.

- **Caddy Integration:** Caddy has native awareness of Tailscale. If the Caddy service allows access to the Tailscale socket, it can request the certificate directly from the local Tailscale daemon.
- **NixOS Configuration:** This typically involves using a specific Caddy build with the caddy-tailscale plugin or ensuring the caddy user has permission to access the Tailscale socket.
  nix services.caddy.virtualHosts."machine.ts.net" \= { extraConfig \= "reverse_proxy 127.0.0.1:8080"; };

This provides a valid public certificate for a private VPN IP, offering the highest level of trust with zero configuration, provided the client is on the Tailnet.[36]

6.2 Dedicated Step-CA

For organizations requiring a full-featured internal PKI (with OIDC identity support, short-lived certificates, and SSH certificates), running a dedicated **Step-CA** instance is an option. NixOS provides a module (services.step-ca) for this purpose.

- **Declarative Setup:** The step-ca configuration can be defined in Nix, though the initialization (generating the initial Root CA and secrets) is often an imperative step (step ca init) performed once, with the resulting artifacts referenced by the module.[39]
- **Caddy Integration:** Caddy can be configured to use this external Step-CA as its ACME issuer (via the acme_ca directive), centralizing trust management for the entire network.[6]

7. Troubleshooting and Operational Nuances

7.1 The "Read-Only File System" Error

A specific failure mode occurs when users attempt to grant port binding capabilities to Caddy using imperative scripts:
Failed to set capabilities on file 'setcap': Read-only file system
This occurs because the script attempts to run setcap on the Caddy binary residing in /nix/store. The solution is to avoid wrapper scripts and instead rely on systemd's capability management, which is handled automatically by the services.caddy module in NixOS.[3]

7.2 DNS Propagation and Resolver Issues

When using the DNS-01 challenge, the ACME client verifies the TXT record propagation.

- **Split-Horizon DNS Conflict:** If the local network uses a split-horizon DNS (where example.com resolves differently internally vs externally), the ACME client might query the internal resolver and fail to see the TXT record pushed to the public DNS.
- **Remediation:** It is best practice to force the ACME client to use a public resolver (e.g., 1.1.1.1) for the validation check. In Nginx/Lego, this is done via the dnsResolver option. In Caddy, this is configured via the resolvers directive within the tls block.[32]

7.3 Debugging TLS Errors

- **SSL_ERROR_RX_RECORD_TOO_LONG:** This error indicates that the client sent an HTTPS request (TLS Client Hello) but the server responded with unencrypted HTTP data. This commonly happens if Caddy is configured for port 80 but the browser forces HTTPS, or if the reverse_proxy directive connects to an HTTPS backend without the transport http specifier.[42]
- **Browser Trust Latency:** After installing a Root CA to security.pki.certificates, a reboot or a restart of the nscd (Name Service Cache Daemon) and the browser is often required for the changes to take effect.

8. Conclusion and Recommendations

The implementation of LAN HTTPS on NixOS demands a shift from imperative thinking to declarative architecture. The data clearly indicates that for the majority of users, **Caddy is the optimal solution**. Its integration of the Smallstep PKI library allows for a seamless "Internal Trust" model (tls internal) that eliminates the complexity of external certificate managers.
Final Recommendations:

1. **For Public Domain Owners (Highest Usability):** Adopt **Strategy B (Split-Horizon)**. Use Caddy with the cloudflare (or relevant) DNS plugin. This provides a "green lock" experience on all devices without client-side configuration.
2. **For Pure Internal Networks (Highest Privacy):** Adopt **Strategy A (Internal Trust)**. Use Caddy with tls internal. Use security.pki.certificates to declaratively distribute the Root CA to NixOS clients, and manually install the root certificate on non-NixOS devices.
3. **For VPN Users:** Leverage **Tailscale** integration for an effortless, zero-config HTTPS

experience across the mesh network.

By adhering to these declarative patterns, the NixOS administrator ensures that the security infrastructure is not only robust but also reproducible, surviving system rebuilds and updates with its trust relationships intact.

Data Comparison Table: DNS Providers for ACME (Partial List)

Provider	NixOS/Lego Support	Caddy Plugin Support	Environment Variable Suffix
Cloudflare	Native	Requires Build (caddy-dns/cloudflare)	CLOUDFLARE_API_TOKEN
Route53	Native	Requires Build (caddy-dns/route53)	AWS_ACCESS_KEY_ID, AWS_SECRET...
OVH	Native	Requires Build (caddy-dns/ovh)	OVH_ENDPOINT, OVH_APPLICATION_KEY
Porkbun	Native	Requires Build (caddy-dns/porkbun)	PORKBUN_API_KEY
Inwx	Native	Requires Build	INWX_USERNAME, INWX_PASSWORD

Works cited

1. How I like to install NixOS (declaratively) - Michael Stapelberg, accessed November 18, 2025, https://michael.stapelberg.ch/posts/2025-06-01-nixos-installation-declarative/
2. configuration.nix: NixOS system configuration specification | Man Page | File Formats - ManKier, accessed November 18, 2025, https://www.mankier.com/5/configuration.nix
3. Using devenv with https : r/NixOS - Reddit, accessed November 18, 2025, https://www.reddit.com/r/NixOS/comments/1mp7u6c/using_devenv_with_https/
4. Setup Caddy as a reverse proxy on NixOS (Part 3 - Ming Di Leom's Blog, accessed November 18, 2025, https://mdleom.com/blog/2020/03/14/caddy-nixos-part-3/
5. localhostd - yoshie - GitLab, accessed November 18, 2025, https://gitlab.com/arkandos/localhostd
6. Declarative insecure self-signed certificates in NixOS - Help, accessed November 18, 2025, https://discourse.nixos.org/t/declarative-insecure-self-signed-certificates-in-nixos/71528
7. Caddy vs Nginx: How Do These Web Servers / Reverse Proxies Compare? - Reddit, accessed November 18, 2025, https://www.reddit.com/r/selfhosted/comments/hur1hx/caddy_vs_nginx_how_do_these_web_servers_reverse/
8. Homelab: Setting up Caddy Reverse Proxy with SSL on NixOS - aottr, accessed November 18, 2025, https://aottr.dev/posts/2024/08/homelab-setting-up-caddy-reverse-proxy-with-ssl-on-nixos/
9. Adding Custom Nginx Options in NixOS - Icewind.nl, accessed November 18, 2025, https://icewind.nl/entry/nixos-add-nginx-options/

10. Caddy - Official NixOS Wiki, accessed November 18, 2025, https://wiki.nixos.org/wiki/Caddy
11. Local valid SSL cert setup - Help - Caddy Community, accessed November 18, 2025, https://caddy.community/t/local-valid-ssl-cert-setup/25277
12. 35 Million Hot Dogs: Benchmarking Caddy vs. Nginx - Tyblog, accessed November 18, 2025, https://blog.tjll.net/reverse-proxy-hot-dog-eating-contest-caddy-vs-nginx/
13. Benchmarking is Hard: Caddy vs Nginx Edition - Patrick D'appollonio, accessed November 18, 2025, https://www.patrickdap.com/post/benchmarking-is-hard/
14. Global options (Caddyfile) — Caddy Documentation, accessed November 18, 2025, https://caddyserver.com/docs/caddyfile/options
15. tls (Caddyfile directive) — Caddy Documentation, accessed November 18, 2025, https://caddyserver.com/docs/caddyfile/directives/tls
16. Automatic HTTPS — Caddy Documentation, accessed November 18, 2025, https://caddyserver.com/docs/automatic-https
17. Install root certificate - Help - NixOS Discourse, accessed November 18, 2025, https://discourse.nixos.org/t/install-root-certificate/25726
18. Caddy, self-signed certificates and Certificate Authorities for web development, accessed November 18, 2025, https://dev.to/migsarnavarro/caddy-self-signed-certificates-and-certificate-authorities-for-web-development-653
19. Caddy - NixOS Wiki, accessed November 18, 2025, https://nixos.wiki/wiki/Caddy
20. Tls internal not working - Help - Caddy Community, accessed November 18, 2025, https://caddy.community/t/tls-internal-not-working/21856
21. Help adding a CA certificate with security.pki.certificateFiles - NixOS Discourse, accessed November 18, 2025, https://discourse.nixos.org/t/help-adding-a-ca-certificate-with-security-pki-certificatefiles/25131
22. Adding a CA Certificate : r/NixOS - Reddit, accessed November 18, 2025, https://www.reddit.com/r/NixOS/comments/1bjn4g0/adding_a_ca_certificate/
23. How to properly add a CA certificate? - NixOS - Reddit, accessed November 18, 2025, https://www.reddit.com/r/NixOS/comments/1f71r3d/how_to_properly_add_a_ca_certificate/
24. Configuration Options — NixOS Manual documentation - GitHub Pages, accessed November 18, 2025, https://nlewo.github.io/nixos-manual-sphinx/generated/options-db.xml.html
25. Help adding a CA certificate with security.pki.certificateFiles - #4 by TLATER, accessed November 18, 2025, https://discourse.nixos.org/t/help-adding-a-ca-certificate-with-security-pki-certificatefiles/25131/4
26. Options - security.pki.cert - NixOS Search, accessed November 18, 2025, https://search.nixos.org/options?channel=unstable\&show=security.pki.certificates\&from=0\&size=50\&sort=relevance\&type=packages\&query=security.pki.cert
27. SSL Certificates - NixOS Wiki, accessed November 18, 2025, https://wiki.nixos.org/wiki/SSL_Certificates
28. Internal SSL error instead of warning for invalid cert - #6 by francislavoie - Help, accessed November 18, 2025, https://caddy.community/t/internal-ssl-error-instead-of-warning-for-invalid-cert/23872/6
29. caddy tls internal not working · Issue #270297 · NixOS/nixpkgs - GitHub, accessed

November 18, 2025, https://github.com/NixOS/nixpkgs/issues/270297
30. ACME - Official NixOS Wiki, accessed November 18, 2025, https://wiki.nixos.org/w/index.php?title=ACME\&mobileaction=toggle_view_desktop
31. Difficult to use ACME/Let's Encrypt with dns.nix · Issue #22 - GitHub, accessed November 18, 2025, https://github.com/nix-community/dns.nix/issues/22
32. ACME with Route53 DNS Challenge Stopped Working - Help - NixOS Discourse, accessed November 18, 2025, https://discourse.nixos.org/t/acme-with-route53-dns-challenge-stopped-working/48378
33. Setting up wildcard Let's Encrypt certificates on NixOS - Carlos Vaz, accessed November 18, 2025, https://carlosvaz.com/posts/setting-up-wildcard-lets-encrypt-certificates-on-nixos/
34. Let's encrypt a service with Taiscale and Cloudflare on Nixos | by stylishavocado - Medium, accessed November 18, 2025, https://medium.com/@stylishavocado/lets-encrypt-a-service-with-taiscale-and-cloudflare-on-nixos-617f20d55fea
35. Nginx reverse proxy with DNS-01 challenge? : r/NixOS - Reddit, accessed November 18, 2025, https://www.reddit.com/r/NixOS/comments/r7y5vy/nginx_reverse_proxy_with_dns01_challenge/
36. Tailscale + caddy certificates not working - Help - NixOS Discourse, accessed November 18, 2025, https://discourse.nixos.org/t/tailscale-caddy-certificates-not-working/61228
37. security/acme-client: auto-renew is failing with Challenge Type http01 · Issue #2394 · opnsense/plugins - GitHub, accessed November 18, 2025, https://github.com/opnsense/plugins/issues/2394
38. Creating private services on NixOS using Tailscale and Caddy | Harsh Shandilya, accessed November 18, 2025, https://drafts.msfjarvis.dev/posts/creating-private-services-on-nixos-using-tailscale-and-caddy/
39. Newcomer to step-ca - Help - NixOS Discourse, accessed November 18, 2025, https://discourse.nixos.org/t/newcomer-to-step-ca/27849
40. Configuring open source step-ca | Smallstep, accessed November 18, 2025, https://smallstep.com/docs/step-ca/configuration/
41. Caddy+Smallstep integration (trusted certs for localhost and internal sites) #3021 - GitHub, accessed November 18, 2025, https://github.com/caddyserver/caddy/issues/3021?timeline_page=1
42. https and Caddy via NixOS and Docker · firefly-iii · Discussion #10835 - GitHub, accessed November 18, 2025, https://github.com/orgs/firefly-iii/discussions/10835

Book Analysis

Ai Editorial Take

This entry stands as a robust treatise on implementing secure, declarative network services within the NixOS ecosystem, a critical aspect of modern 'Future-proofing'. The deep dive into Caddy, Nginx, and various HTTPS strategies (Internal Trust, Split-Horizon, Tailscale, Step-CA)

is exceptionally well-researched and directly addresses complex architectural challenges. The narrative skillfully interweaves AI collaboration, demonstrating a powerful methodology for human-AI partnership in system administration. This methodology for managing infrastructure as code with explicit, verified steps is an important blueprint for developing resilient and trustworthy digital systems.

Content Potential And Polish

- **Core Strengths:**
- Comprehensive overview of NixOS configuration modularization for services, demonstrating a best-practice architectural shift.
- Detailed breakdown of Caddy and Nginx for HTTPS on LAN, including comparative analysis, pros/cons, and specific NixOS implementation details for each strategy (Internal Trust, Split-Horizon).
- Clear articulation of the 'dry-run' discipline for validating AI-generated NixOS configuration changes, emphasizing the human-AI partnership in system administration.
- Narrative flow that connects initial system recovery and AI collaboration with complex architectural refinement for network services.
- Thorough research and extensive citations backing the technical recommendations, enhancing credibility and depth.
- **Suggestions For Polish:**
- Elaborate on the practical, step-by-step instructions for distributing the internal Root CA to various client operating systems (Windows, macOS, Android, iOS) when using Caddy's tls internal strategy.
- Provide concrete, fully functional NixOS code snippets for a complete Caddy setup that includes multiple virtual hosts and demonstrates a DNS challenge configuration (with appropriate placeholders for sensitive API keys).
- Expand the troubleshooting section to include more common HTTPS-related errors specific to NixOS, such as issues with certificate paths, permissions, or Nginx's setcap conflicts.
- Include a summary block or diagram that visually represents the configuration.nix, packages.nix, and services.nix files after the refactoring, showing the cleaned 'Motherstone' and the modular 'Town Square'.

Next Step Prompts

- Generate a detailed NixOS configuration snippet for implementing Caddy with tls internal and step-by-step instructions for exporting and importing the Root CA into various client operating systems (Windows, macOS, Linux, iOS, Android).
- Develop a comprehensive, step-by-step guide for configuring Caddy with a DNS-01 challenge (e.g., Cloudflare) in NixOS, including custom package builds for DNS plugins and secure secret management via systemd's EnvironmentFile.