ChatGPT

Designing a Netplan-Based Networking Module for Cockpit

Netplan Configuration Files and the Multi-File Approach

Netplan uses YAML files (in `/etc/netplan/`) to configure network interfaces on Ubuntu. Multiple YAML files can coexist and are **merged** by netplan at runtime. The merge follows lexicographical order of filenames: later (alphabetically higher) files **add to or override** settings from earlier files [1]. In practice, a file named `01-custom.yaml` would override settings from `00-installer-config.yaml` if they define the same keys. Netplan's merge rules are as follows [2]:

- **Scalars/booleans**: A later file's value replaces the earlier value.
- **Sequences (lists)**: Later entries are **concatenated** to earlier ones (appended rather than replacing).
- **Mappings (dicts)**: Merged recursively – keys present in the later file override or add to earlier mappings.

Because of these rules, **care must be taken** to avoid unintentional merges. For example, if two files set DNS `nameservers` for the same interface, the final DNS list will combine entries from both files instead of one replacing the other [3]. This can lead to duplicate or unintended addresses/routes if an interface is defined in multiple files. **Best practice** is to minimize overlapping definitions across files – ideally, define each interface (or each setting) in only one file to avoid merge conflicts.

Using a **multi-file setup** is beneficial for separating base system config from custom changes. Ubuntu's installer typically creates a base config file (e.g. `00-installer-config.yaml`) with initial network settings. You can then add custom files like `50-cockpit.yaml` or `99-custom.yaml` for new interfaces or overrides. Ensure your custom file name is lexicographically last so that your settings have highest priority [4]. Keeping the number of files low (perhaps one base file and one custom file) will make management simpler and reduce confusion. Each additional file increases complexity in how settings merge, so only use multiple files when necessary. In most cases a **single consolidated file** (or a base+one override file) is sufficient for a "production-grade" setup, as this avoids tricky merge behavior while still allowing separation of system defaults from user modifications.

Managing Baseline vs. Custom Netplan Configurations

Baseline Config (System-Provided): The `00-installer-config.yaml` (or similar) contains the default network config (usually your primary NIC configured via DHCP or static IP from installation). This file should be parsed and its contents displayed in the UI as the initial state of those interfaces. It's wise to **preserve the baseline file** so that if something goes wrong, you know the original working config. In a multi-file approach, the baseline can remain untouched for reference, while custom changes live in new files. However, remember that if an interface is defined in both the base and a custom file, netplan will merge or override settings as described above.

Custom Config Files: For any new interfaces (VLANs, bonds, etc. not in the baseline), you can safely create a new YAML file to define them. Since these are new definitions, they won't conflict with the base file. The netplan generator will simply union these configurations. For example, you might have `01-xavs-network.yaml` containing all user-created interfaces and their settings, applied after the `00-installer-config.yaml`. By naming it with a higher number, you ensure it overrides any overlapping keys (if any) from the base [4].

Editing Existing Interfaces: This is the tricky part. If the user wants to edit a **system-defined interface** (e.g. change the IP or DHCP on the primary NIC), you have two options:

- **Edit the Base File Directly:** Simplest method – modify `00-installer-config.yaml` with the new settings. This ensures there's only one definition of that interface. It avoids merge issues (since no second file defines the same interface) and keeps the final config clear. If you choose this, make a backup of the original file in case a rollback is needed. Also, clearly mark in the UI that this is a critical change to a system interface. After editing, run `netplan generate` to validate syntax and `netplan try` to test (more on this below) before applying permanently.

- **Override via a Custom File:** Alternatively, leave the base file untouched and create entries in your custom file for that interface. In theory, netplan will override scalar settings (like `dhcp4` or a gateway) with your new values [1]. **However, be cautious**: some settings are lists (addresses, DNS, routes) which will **merge** rather than replace [2]. For instance, if the base file has an IP address or DNS server and your custom file adds another, the interface may end up with both IPs or both sets of DNS. This can lead to "lost" or duplicate configurations when you expected a replacement. An example is trying to override DNS: one file's DNS servers ended up **cumulated** with the other's [3]. Netplan does not have a built-in "replace vs merge" switch for config files – it always merges as per the rules. The only way to truly override a list is to **remove the original entry** (which you can't do from another file unless you delete the original file).

Therefore, if taking this approach for existing interfaces, you often end up effectively **replacing the base file anyway**. One strategy is to copy the interface's full config from the base into your custom file (with desired changes), then either comment out or delete that interface's stanza in the base file (or rename the base file so netplan ignores it). This ensures the interface is defined only once (in the custom file). Essentially, you'd be consolidating that interface's config into the new file. This achieves the override without merging conflicts, at the cost of not using the base file for that interface anymore. In summary, **do not define the same interface in two files** unless you deliberately want merged settings; to avoid loss of IPs or routes on VLANs and other interfaces, keep each interface's config in a single YAML file context.

Given the above, a robust production approach is: **use one netplan YAML file for all active config**, if possible. You can still preserve the original as a backup. In practice, your Cockpit module could read `00-installer-config.yaml`, incorporate any user changes or additions into an internal data model, and then write out **either** an updated `00-installer-config.yaml` **or** a new `99-xavs-network.yaml` that contains the entire intended configuration (including both original and new interfaces). Many admins simply have one `/etc/netplan/config.yaml` in the end. Fewer files means fewer merge surprises. Just remember to remove or disable any superseded files so they don't linger and confuse netplan. (For example, if you switch to one combined file, rename `00-installer-config.yaml` to something like `00-installer-config.yaml.bak` so that netplan doesn't read it.)

Discovering and Displaying Network Interfaces

Before making changes, the module should enumerate all network interfaces and their properties. This can be done by parsing system commands like `ip link show` or `ifconfig -a` to list interfaces, and `ip addr show` to get IP assignments. Cross-reference this with netplan's config to determine which interfaces are managed and how:

- Interfaces present in netplan YAML (under `ethernets`, `vlans`, `bonds`, `bridges`, etc.) are "managed" and have known configurations.
- Interfaces shown by `ip link` that are **not** in any netplan file are currently unmanaged (or managed by another system like NetworkManager, if applicable). The UI can present those as available for configuration.

For each interface, determine its type: - Physical NICs usually appear under the `ethernets:` section in YAML (or `wifis:` for wireless). - VLAN interfaces appear under a `vlans:` section (with an `id` and parent `link`). - Bonds under `bonds:`, Bridges under `bridges:`, and so on. - Some interfaces (like the loopback `lo`) might appear under `ethernets` as well. Loopback is always present but usually only loopback addresses are configured on it if needed.

The Cockpit UI should display all interfaces (physical and virtual) and their key info: IP addresses, DHCP/static, up/down status, etc. This likely involves parsing the YAML files: you can use a YAML library or `netplan --debug generate` to see the parsed output. Ensure the UI clearly distinguishes different interface types (e.g., maybe icons or labels for "VLAN", "Bridge", "Bond" etc.), since their configuration options differ.

Configuring Interface Types via Netplan

When the user creates or edits an interface through the UI, the module should update the netplan YAML accordingly. Here's a breakdown of how to handle each major interface type in netplan:

- **Physical Ethernet Interfaces:** These are defined under the `network.ethernets` section (for wired NICs). If using DHCP, simply set `dhcp4: true` (and `dhcp6: true` if IPv6 via DHCP) [5]. For static configurations, specify one or more addresses and a default gateway. For example, a static IPv4 config might look like:

```
network:
  version: 2
  renderer: networkd
  ethernets:
    enp3s0:
      addresses:
        - 10.10.10.2/24
      nameservers:
        search: [mydomain, otherdomain]
        addresses: [10.10.10.1, 1.1.1.1]
      routes:
```

```
      - to: default
        via: 10.10.10.1
```

In this example, `enp3s0` gets a static IP `10.10.10.2/24`, with DNS servers and a default route via 10.10.10.1 [6]. The UI should collect these fields (IP address/prefix, gateway, DNS, etc.). If multiple IPs are needed on one NIC, netplan supports a list under `addresses:` (and you can give labels to secondary addresses) [7] [8], but that can be an advanced scenario. Start with single-address support, and possibly allow adding extra addresses via the UI as needed.

Note: If the physical NIC is part of a bond or bridge (see below), **do not assign an IP to the physical NIC itself** – the IP should be on the bond or bridge. In such cases, set the physical `dhcp4: no` and no addresses (or omit the `addresses` key) so it doesn't try to get an IP on its own. Often one also sets `optional: true` on slave interfaces so that boot doesn't wait indefinitely for them [9]. The UI can enforce this by automatically marking member interfaces accordingly.

- **VLAN Interfaces:** VLANs are configured under `network.vlans`. Each VLAN needs a unique name (this becomes the interface name in the system, e.g. "vlan10" or you can name it like "enp3s0.10" but YAML uses a key as the name). In YAML you specify the VLAN ID and the parent link. For example:

```
vlans:
  vlan15:
    id: 15
    link: mainif
    addresses: [ "10.3.99.5/24" ]
    nameservers:
      addresses: [ "127.0.0.1" ]
      search: [ domain1.example.com ]
```

Here, `vlan15` is a VLAN interface with tag 15 on parent interface `mainif` (which would be a physical NIC defined elsewhere) [10] [11]. It has a static IP 10.3.99.5/24 and some DNS servers. If DHCP is desired on a VLAN, use `dhcp4: true` instead of static addresses. The key is to ensure the **parent interface exists** (in this case `mainif` must be defined under `ethernets` with at least `dhcp4: no` and no conflicting IP). Typically, if you configure a VLAN on a NIC, the NIC itself might either have no IP or have only untagged network IPs. Netplan is smart enough to bring up the parent link when a VLAN on it is configured.

Common pitfalls to avoid: - Not specifying the `addresses` or `dhcp4` for the VLAN (resulting in it coming up with no IP). - Assigning the IP to the wrong entity (e.g., giving the parent NIC the IP when it's supposed to be on the VLAN). The UI should clarify that if you want an IP on a tagged network, you must create a VLAN interface for that tag and assign IP to it, not to the base NIC (unless using the untagged network). - Use descriptive names for VLAN interfaces in the YAML to avoid confusion, especially if there will be many. (Netplan examples use names like "vlan10", but you could also use "eno1.10" as the key; the name outside can be arbitrary, though by convention it often reflects the VLAN).

- **Bonded Interfaces (Link Aggregation):** Bonds are configured under `network.bonds`. You define a bond interface name and list the member interfaces. Do **not** give the member NICs their own IP – the bond itself will have the IP/DHCP. Example:

```
bonds:
  bond0:
    interfaces: [enp3s0, enp4s0]
    dhcp4: yes
    parameters:
      mode: active-backup
      primary: enp3s0
```

This creates `bond0` bonding `enp3s0` and `enp4s0` in active-backup mode (so one is primary, the other fails over) [12]. If you want a static address on the bond, replace `dhcp4: yes` with an `addresses:` list and `gateway4` or `routes` as needed (similar to a physical NIC). There are various bonding modes (balance-rr, 802.3ad (LACP), etc.) – the UI can provide a dropdown for mode and additional parameters (like LACP rate, miimon frequency). The netplan docs examples show how to set those parameters [13]. At minimum, support active-backup and 802.3ad modes initially, since those are common.

Important: The member interfaces (e.g., `enp3s0`, `enp4s0`) should be defined under `ethernets` with `dhcp4: no` and typically `optional: true` [9]. This prevents them from trying to configure themselves independently and allows the system to boot even if they're not active. Netplan will enslave them to the bond when bringing `bond0` up. If the baseline config had those NICs configured, you'll need to remove or adjust that when bonding them (e.g., if `enp4s0` had a config in the base file, you must remove that config when adding it to a bond). The Cockpit module should warn if the user tries to bond an interface that's already configured with an IP – it should either remove that IP or abort until they remove it, to avoid duplicate config.

- **Bridge Interfaces:** Bridges are configured under `network.bridges`. A bridge behaves like a virtual switch that can have one or more member interfaces (ports), and optionally an IP if the bridge itself should have a layer-3 presence. To create a simple bridge with one physical NIC:

```
bridges:
  br0:
    interfaces: [enp3s0]
    dhcp4: yes
```

This means `enp3s0` will become a port in `br0`, and `br0` will get an IP via DHCP [14]. If static, you'd put addresses and routes under `br0` just like any other interface. The member interfaces (`enp3s0` in this case) should be defined under `ethernets` with `dhcp4: no` (since they won't have a separate IP) – netplan will automatically handle bringing them up as part of the bridge. You might also mark them optional. In a bridge, typically the bridge is the one that carries the IP, and the slaves have none.

Bridges can also include VLANs or bonds as members. For instance, you might have a bond of two NICs, and then put that bond into a bridge, or have a VLAN interface as a member of a bridge (for more advanced networking scenarios like trunk ports to a VM hypervisor). Netplan supports that (e.g., `interfaces: [ vlan15 ]` under a bridge, with `vlan15` defined as above [15] [16]). For phase 1, you might keep it simple: allow bridges with physical NICs as members; later you can expand to bridging VLANs if needed.

Ensure the UI prevents common mistakes, such as bridging an interface that already has an IP config (the IP should be moved to the bridge). Similar to bonds, if a NIC is in the baseline config and the user wants to bridge it, you'll need to transfer that config to a new bridge interface. Possibly automate: e.g., user selects "bridge enp3s0", the module could create br0, put enp3s0 as member, and move enp3s0's IP settings to br0 in the YAML.

- **Static Routes:** Netplan allows specifying static routes per interface under a `routes:` list. For each route, you provide at least `to` (destination subnet or `default` for 0.0.0.0/0) and `via` (gateway IP). Optionally, `metric:` can set the route priority and `table:` can specify a routing table (for policy routing, see phase 2). Example of adding an extra static route:

```
ethernets:
  enp3s0:
    addresses: [10.0.0.10/24]
    routes:
      - to: 10.5.0.0/16
        via: 10.0.0.1
        metric: 50
```

This would route traffic to 10.5.0.0/16 via 10.0.0.1. For a default gateway, netplan uses either the `gateway4` / `gateway6` shortcut or a route entry with `to: default` [17] – both result in a default route. If multiple interfaces have default routes, **use metrics** to avoid conflicts: the lower metric wins as the primary route [18]. Netplan by default assigns metric 100 to DHCP-obtained routes, so if you add a static default route manually, you might choose metric 50 to make it primary, or 200 to make it secondary, etc [18] [19]. The UI should expose setting static routes for each interface (maybe a sub-form to add routes: destination, gateway, metric).

Preserving existing routes: If the base config or current state has custom static routes, make sure to include them in the netplan YAML. Routes aren't magically preserved unless they're in the config. So if a user had a static route configured outside of netplan (or via cloud-init), you'll want to capture that and add it to the YAML; otherwise, when you apply, that route will disappear. Similarly, if you reconfigure an interface that had a default gateway, ensure the new config still has a default route (or another interface does), or the system might lose internet connectivity. The module could warn if no default route remains.

Applying Changes Safely (Generate, Try, Apply)

Making network changes can be disruptive, so a careful apply workflow is essential:

1. **Generate & Validate:** When the user submits changes (add/edit interface), first update the YAML files on disk (or generate the new consolidated YAML). Then run `**netplan generate**` to have netplan parse the files and generate the low-level configuration. This will catch YAML syntax errors or schema issues. If `netplan generate` reports errors, present those to the user to fix (e.g., indentation errors, invalid keys, etc. – common if the YAML is malformed [20]). Only proceed to apply if generation succeeds without errors.

2. **Trial Apply (No Lockout):** Use `**netplan try**` for a safe test apply. The `netplan try` command applies the new configuration **temporarily** and then waits for confirmation [21]. By default it gives 120 seconds to confirm, otherwise it auto-rolls back to the previous config [22] [23]. This is extremely useful in a Cockpit scenario: if the user is connected over the network (e.g., via the very interface they are reconfiguring), a wrong change could cut off connectivity. By doing a trial, netplan will revert automatically if the user doesn't confirm.

The challenge is that `netplan try` expects a confirmation (you normally press **Enter** in the console). In a web UI, you can't easily press enter on the server side, but you *can* detect if the connection stays up. One approach: after calling `netplan try`, attempt to ping or AJAX-call the server from the UI. If responses are coming, the network is up. You could then run `netplan apply` (perhaps via a background job or ask the user to confirm by clicking a button). If the network goes down, the user won't be able to confirm – but that's fine, because after 120 seconds, netplan will rollback automatically [22]. You should inform the user of this timeout: e.g., "Applying new settings... if you do not confirm within 2 minutes, changes will revert." Possibly provide a "Confirm now" button that triggers the confirmation. Netplan allows non-interactive confirmation by sending a signal or running `netplan apply` while `try` is in progress (or simply invoking `netplan try --timeout=10` with a short timeout for quick tests).

In summary, **always use `netplan try` for critical changes**. This prevents permanent lockouts on remote systems [24]. For less critical changes (adding a secondary interface that doesn't affect your connectivity), it might be acceptable to do a direct apply – but using `try` consistently is a good safety net.

1. **Permanent Apply:** If the test is successful (i.e., user confirmed or the network is verified), finalize with `**netplan apply**`. This writes the config and brings up the interfaces permanently. Unlike `try`, `netplan apply` doesn't wait or rollback – it just applies immediately. So do this only after you're confident in the config (for example, after a successful `try`). In some implementations, after a `try` is confirmed, it actually already applied the config, so an explicit `apply` might not be needed; however, doing an `apply` ensures the configuration is definitely applied and saved (netplan will re-read the same files and apply again, which should be a no-op if nothing changed, or will finalize any pending changes if needed).

2. **Error Handling:** If `netplan try` or `apply` fail (e.g., netplan finds an invalid configuration that `generate` didn't catch, or a runtime error occurs bringing up interfaces), capture that error output and display to the user. Netplan is usually good about reporting what went wrong (like "Failed to bring up bond0: some interface missing" etc.). In case of failure, the module should rollback the YAML changes if necessary (for example, if you removed the base file or something, you might want to restore it). Generally, if `apply` fails, the system may be left in a partially configured state, so test thoroughly. This is another reason `try` is useful: it will rollback automatically if something fails within the timeout.

3. **Graceful Restart:** After applying, it might be wise to refresh the UI's view of interfaces (since new ones might have appeared or old ones went down). You can run `ip addr` again or `netplan list` to update the displayed information.

Removing and Disabling Interfaces

Your module should also handle interface deletion (or deactivation) cleanly:

- **Deleting a Virtual Interface:** For interfaces like VLANs, bonds, bridges (which are created via netplan), removing their config from the YAML and applying will cause them to be torn down. To delete, remove the stanza from the netplan YAML file(s) (or mark it as deleted in your data model and rewrite files without it). Then run `netplan apply`. Netplan will remove the IP addresses and routes associated with that interface. In the case of a bond or bridge, it will release the member NICs back to standalone state (though those NICs might then be down if they have no config).

Additionally, to completely remove the interface device from the runtime system *immediately*, you can use the `ip link delete <iface>` command for VLANs or bridges. For example, `ip link delete vlan15` will remove that VLAN interface from the kernel [25]. Netplan apply usually handles this, but if you want to be sure or do it instantly before apply (or if netplan doesn't fully remove it until reboot), the `ip link delete` is a good step. **Do not** try to delete physical interfaces (you cannot remove a physical NIC device via software; you can only down it or ignore it).

- **Deleting a Physical Interface's Config:** You obviously can't remove a physical NIC from the system, but you can remove its network configuration so that it's "unused". This might be done if, say, the user wants to disable a secondary NIC or move it from static to completely unmanaged. In netplan, you'd take out its entry from YAML. After applying, that NIC will have no IP (and will likely be down, or at least not have an address). If that NIC was part of a bond/bridge, handle accordingly (removing it from the bond's interface list etc.).

Be mindful of dependencies: If the user tries to delete an interface that others rely on (e.g., deleting a bridge that has slaves, or a physical NIC that has VLANs), the module should either prevent it or cascade-delete the dependents. Ideally, enforce a hierarchy: you cannot delete a physical device's config if there are VLANs on it (require deleting those first), cannot delete a bond without first removing or reassigning its slaves, etc. Provide warnings in such cases.

- **Interface Up/Down toggling:** The UI might have "Enable/Disable" or "Up/Down" buttons. There are a couple ways to implement this:
- **Temporarily (runtime) down/up:** Use `ip link set <iface> down` (or `up`) to change the state immediately. This does not change the persistent config, so on reboot or `netplan apply` it will revert to whatever the config says. This is akin to just shutting an interface without removing its config. It could be useful for troubleshooting (e.g., temporarily disable a NIC). If you do this, the UI should show the state (you can get admin state from `ip link`). The user needs to know that after a reboot or if they reapply netplan, the interface will come back up because its config is still there.
- **Persistently disabling:** Netplan doesn't have an explicit "disable interface" toggle in YAML. To persistently keep it down, you effectively remove its config (as described above). Another hack is to set an interface to **unmanaged by netplan** by not listing it at all in YAML – then nothing will bring it up (but something else might, e.g., NetworkManager if running). Typically, removing it from netplan is the way to disable. If you want to preserve the config for later, you could comment it out in the YAML (netplan ignores YAML comments) or maintain a separate store of "disabled interfaces" that could be re-enabled later. But netplan itself has no notion of an interface that is defined but

administratively down – except if you define it with no addresses and no DHCP, it will be up (in layer2) but with no IP.

In summary, for a persistent disable, treat it as deleting the config (with the option to add it back later). For a momentary toggle, use `ip link`. The module could support both, but clarify to the user what each means.

- After deleting or disabling, **clean up routes**: Any static routes tied to that interface should be removed from YAML too. Netplan will remove them upon apply when the interface config is gone. If the interface was providing the default route and it's removed, ensure another interface has a default route if needed (or the system will lose default gw). Perhaps warn: "You are removing the interface that provides the default gateway. This may cut off external network access."

- As always, use `netplan try` if deleting a critical interface (like if they delete an interface they're connected through) to allow auto-revert. However, deletion is tricky to revert (netplan try might bring it down and then revert by re-adding config). It should work, but definitely test that scenario. If not confident, require the user to confirm via console before deleting the mgmt interface.

Warnings and Safety Checks for Critical Interfaces

Network changes can have immediate consequences. The Cockpit module should integrate **warning prompts** for operations that could disrupt connectivity:

- **Editing the management interface:** Detect if the interface being modified or brought down is the one currently providing Cockpit access (you might detect this by matching the interface IP with the client's IP or default route interface). If yes, present a bold warning: *"You are about to modify the interface you are connected through. This could disconnect you."* In such cases, **definitely use** `netplan try` and instruct the user to be prepared to lose connection temporarily. Encourage them to ensure they have out-of-band access (like a VM console or physical access) just in case.

- **Deleting critical routes:** If a user removes a default route or all IP addresses from an interface, warn that the system might become unreachable on that network. For instance, deleting the only interface with internet connectivity.

- **Bringing down important links:** If they down a bond that carries multiple VLANs or similar, multiple networks might go down at once. Warn accordingly.

- **General Confirmation:** It's good UX to have a confirmation dialog for destructive actions like delete or power off an interface. Use clear language like "Are you sure you want to delete interface VLAN 15 (vlan15)? This will remove its IP configuration and the interface will be brought down."

- **Use of `optional: true`:** For interfaces not crucial for boot (like auxiliary NICs, or VLANs on a port that might not always be present), consider setting `optional: true` in YAML [26]. This prevents systemd-networkd from waiting 2 minutes at boot for that interface to come up. However, do **not** mark the primary interface as optional, or you could delay network availability. Netplan automatically marks DHCP interfaces optional if they're not found, but explicitly doing so for nonessential ones can improve boot time. The module could toggle this flag based on a user setting like "Skip wait at boot".

By implementing these warnings and using netplan's safety features, you can greatly reduce the chances of users inadvertently cutting off their server's network. The goal is that any change made via the UI either succeeds or safely rolls back without requiring a manual fix.

Future Enhancements (Phase 2: Advanced Networking)

Once the basic management of physical NICs, VLANs, bonds, bridges, and static routes is solid, you can plan for more advanced features in Phase 2:

- **Policy-Based Routing (Routing Rules):** Netplan supports advanced routing policy rules via the `routing-policy` YAML key. This corresponds to the Linux policy routing (IP rule) often configured in systemd-networkd's `[RoutingPolicyRule]` sections. For example, to force traffic from a certain subnet out a specific routing table, you might add in YAML:

```
routing-policy:
  - from: 192.168.3.0/24
    table: 101
```

along with a static route in that table (e.g., `routes: [ { to: 192.168.3.0/24, via: 192.168.3.1, table: 101 } ]`) [27] . The netplan example [28] shows two interfaces each with their own routing table and a policy rule to direct traffic from each subnet into its respective table. Implementing this in the UI means exposing fields for advanced users: e.g., "Source-based routing" where they can specify a source subnet and a routing table number, and add corresponding routes. It's advanced, so likely hide behind an "Advanced settings" toggle. **Important:** Each routing-policy rule should have a unique priority if multiple rules exist, to define ordering (Netplan auto-generates priorities if not given, but it's recommended to set them) [29] . Phase 2 can handle these details. For now, design the data model such that routing rules can be added later.

- **Additional Virtual Interfaces:** There are other interface types netplan can manage: **tunnels** (SIT, GRE, IPIP, WireGuard, etc.), **MAC VLANs**, **VXLAN**, etc. For instance, netplan's `network.tunnels` section can configure a GRE or WireGuard tunnel [30] [31] . If your use case requires (perhaps not immediately), you can plan to support adding a "Tunnel" interface in the UI (with parameters like local/remote addresses, keys for WireGuard, etc.). Similarly, support for **PPP or modems** could be considered if needed (though those might be out of scope for server management in Cockpit).

- **Wi-Fi Management:** If this module will ever run on systems with Wi-Fi, netplan can configure Wi-Fi under `wifis:` with SSID and passphrase [32] . It might not be a priority for a server module, but worth keeping in mind.

- **Netplan D-Bus API:** Netplan has a D-Bus API for programmatic config (in newer versions). In the future, instead of writing YAML files directly, the module could interact with netplan via D-Bus to apply changes. This could avoid manual file parsing/writing and might give transactional abilities. However, that's optional – editing YAML and calling netplan CLI is perfectly fine and common.

- **UI/UX Improvements:** As more features are added, ensure the UI remains user-friendly. Possibly implement profiles or backups (e.g., allow user to revert to a previous known-good config easily). Logging of changes could also be useful.

Throughout phase 2, maintain the guiding principles from phase 1: **avoid conflicts, preserve existing config unless intentionally changed, and safeguard connectivity with warnings and `netplan try`.** Each new feature (like routing rules) should be tested in a safe environment to see how netplan applies them and if any caveats arise.

Conclusion

By following a clear structured approach, the XAVS Networking Cockpit module can reliably manage network interfaces using netplan:

- Use a minimal number of netplan YAML files to avoid complexity, leveraging netplan's lexicographic override behavior wisely [1] [4].
- Parse and present existing configuration, separating system (installer) config from user-defined config, but ultimately consolidating changes so that each interface's settings come from a single source to prevent unexpected merges.
- Support key interface types (physical NICs, VLANs, bonds, bridges) with proper netplan YAML schemas, ensuring IPs and routes are assigned to the correct layer (e.g. VLAN vs physical, bond vs member) [33] [10]. Maintain network stability by preserving important settings like routes and only removing them when intended.
- Implement robust apply logic using **netplan generate** (validate) and **netplan try** (safe apply with rollback) [22]. This will make the module *production-grade*, as changes can be made with confidence that a mistake can automatically revert, preventing downtime.
- Provide a user-friendly GUI experience with dynamic forms and modals for editing/adding interfaces, with clear warnings for critical operations. For example, notify if they're about to lose connectivity and require explicit confirmation for such actions.
- Keep an eye toward future enhancements (policy routing, tunnels, etc.), designing the data model and code structure to extend easily in Phase 2, once the core is stable.

With this approach, the Cockpit networking module will turn netplan's powerful (but sometimes intricate) configuration system into a manageable, safe, and effective UI-driven network management tool. It will effectively bridge the gap between low-level YAML configs and user-friendly control, suitable for production environments where reliability is paramount.

---

[1] [2] configuration - Netplan config files overwriting each other - Ask Ubuntu
https://askubuntu.com/questions/1339062/netplan-config-files-overwriting-each-other

[3] ubuntu 20.04 - How to get netplan to replace sections instead of merge? - Stack Overflow
https://stackoverflow.com/questions/71839801/how-to-get-netplan-to-replace-sections-instead-of-merge

[4] [20] [21] [25] How to Install & Configure CSF for cPanel/WHM
https://netrouting.com/knowledge_base/configuring-bonding-on-ubuntu-with-netplan/

5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 26 27 28 30 31 32 33 Examples - Netplan documentation
https://people.ubuntu.com/~slyon/netplan-docs/examples/

22 23 24 Ubuntu Manpage: netplan-try - try a configuration, optionally rolling it back
https://manpages.ubuntu.com/manpages/jammy/man8/netplan-try.8.html

29 YAML configuration - Netplan documentation
https://netplan.readthedocs.io/en/latest/netplan-yaml/