ChatGPT

Informe de Auditoría Remoto (A)

- **Servicios activos:** Actualmente, bajo la política `solo_madre` (modo por defecto), sólo están corriendo Madre (puerto 8001) y Redis (6379) [1]. El conmutador (Switch) y otros servicios opcionales están detenidos por política; se pueden iniciar manualmente bajo demanda (en auditoría se comprobó exitosamente encender Switch con `docker compose up switch` [2]).
- **Base de datos (DB):** La BD unificada está intacta. Las comprobaciones SQLite (`PRAGMA quick_check, integrity_check`) han resultado sin errores [3] y el `DB_SCHEMA_v7_FINAL.json` muestra 71 tablas y 1.15M de filas. Se realizaron copias de seguridad (2 actuales, 23 archivadas) como registra el scorecard [3]. No hay violaciones de clave foránea conocidas.
- **Madre (Power Manager):** Madre responde al endpoint `/health` y acepta comandos de política y ventanas. Se verificó que aplicar la política `solo_madre` detiene correctamente servicios no esenciales [4] y que iniciar servicios autorizados (e.g. Switch) funciona conforme lo esperado [2]. El estado (`/madre/power/state`) refleja `policy: "solo_madre"` y sólo los servicios canon (madre+redis) activos.
- **Switch (Adaptive Engine):** Inicialmente detenido por política, pero arrancó correctamente on-demand. Su endpoint `/health` responde cuando se levanta. Se comprobó que el enrutamiento de chat a Switch funciona: al invocar `/switch/route` el servicio devuelve respuesta HTTP 200 y registra un evento en la tabla `routing_events` [5] [6]. (En auditoría de flujo integrado se obtuvo respuesta de Switch y se almacenó registro en BD [7] [6]).
- **Chat Operador (tentáculo_link):** El endpoint `POST /operator/api/chat` está activo. Se confirmó que recibe mensajes y aplica correctamente rate-limit, tamaño máximo, y responde. Sin embargo, se identificó un error clave: el cliente frontend usaba **URL fija** `http://localhost:8000`, lo cual provocaba **NetworkError** al servirse la UI en otro host/origen [8] [9]. Ésta fue la causa exacta del error de red en la UI Chat (código frontend `BASE_URL = 'http://localhost:8000'`) [8] [9]. Se corrigió luego a URL relativa (`VITE_VX11_API_BASE_URL ?? ''`) [9].
- **UI Operador:** El frontend (en `/operator/ui/`) estaba construído correctamente pero romperá en entornos distintos a localhost debido al `BASE_URL` hardcodeado. En pruebas manuales se sigue el procedimiento indicado en el inventario de pasos [10] [11]. Actualmente, tras la corrección del `BASE_URL`, la UI carga sin NetworkError y las peticiones `/operator/api/*` son relativas (mismo origen) [9] [11].
- **Derivas ("drift") vs. cánon:** No se detectaron módulos inesperados en el sistema (la auditoría canónica de filesystems no arrojó rutas extra). El script de "drift" en tierra indica Fs correctos (no existen carpetas prohibidas como `hermes` o `shubniggurath_legacy`). El número de tablas (71) y servicios coincide con el canon actualizado. La BD cumple las invariantes (chequeos ok) y la política canónica de retención se aplica vía `/madre/power/maintenance/post_task` [3]. En resumen, no hay desviaciones críticas respecto al estado canónico (el "drift" es cero).

Diagramas como Contrato (B)

Flujo de Chat del Operador (Mermaid):

```
graph LR
  Browser[Cliente Operator UI] -->|POST /operator/api/chat|
Tentaculo[Tentáculo Link]
  Tentaculo --> Cache[Redis (cache de chat)]
  Tentaculo --> RateLimit[Redis (rate-limit)]
  Tentaculo -->|Intentar Switch (6s)| Switch[Servicio Switch (CLI Copilot)]
  Switch -->|Éxito| Tentaculo
  Switch -->|Timeout o falla| LLM[LLM local degradado]
  LLM --> Tentaculo
  Tentaculo -->|Respuesta (JSON)| Browser
```

Este diagrama muestra que la interfaz de usuario envía la solicitud de chat al gateway (`tentáculo_link`), que primero verifica cache y rate-limit. Luego intenta enrutar la pregunta al Switch (tiempo máx. 6s). Si Switch responde exitosamente, retorna la respuesta al UI. Si Switch falla o no responde, entra la **degradación local**: un LLM local genera respuesta degradada (2s) y se reenvía al UI. (Eventualmente, si se habilitase `DeepSeek`, podría intentarse, pero por defecto está desactivado en P12 [12]).

Flujo de Respaldo Degradado (LLM):

```
flowchart TD
  UserQuery[Solicitud de chat (UI)] --> Tentaculo
  Tentaculo --> SwitchChk{Switch activo?}
  SwitchChk -- Sí --> SwitchProc[Procesar en Switch]
  SwitchChk -- No --> LLMChk{LLM local habilitado?}
  LLMChk -- Sí --> LLMProc[Procesar con LLM local (degradado)]
  LLMChk -- No --> DeepSeekChk{DeepSeek habilitado?}
  DeepSeekChk -- Sí --> DeepSeekProc[Procesar en DeepSeek API]
  DeepSeekChk -- No --> ErrorResp[Responder “Serviço no disponible”]
  SwitchProc --> Browser
  LLMProc --> Browser
  DeepSeekProc --> Browser
  ErrorResp --> Browser
```

Este flujo ilustra la cadena de **fallback** del chat: tras intentar Switch, se pasa a LLM local degradado; si éste también estuviera deshabilitado (o fallase), se consideraría DeepSeek (aunque en P12 es opt-in sólo para laboratorio) y, al final, un modo degradado estático si todo falla [13] [12].

Flujo de Post-task / Mantenimiento (Madre):

```
sequenceDiagram
  actor System
  System->>Madre: POST /power/maintenance/post_task
  Madre->>DB: PRAGMA quick_check, integrity_check, foreign_key_check
  DB-->>Madre: chequeos OK
  Madre->>DB: Regenerar esquemas (DB_SCHEMA_v7_FINAL, DB_MAP_v7_FINAL)
  Madre-->>System: Resultado del audit (tablas, rows, tamaño) [3]
```

```
  Madre->>Backups: .backup rotación
  Backups-->>Madre: respaldos actualizados (2 activos, 23 archivados) [3]
```

Tras la ejecución de un conjunto de tareas, el endpoint `/madre/power/maintenance/post_task` aplica la retención de datos y valida la integridad BD [3]. Se efectúan los `PRAGMA quick_check`, se regeneran los artefactos DB_SCHEMA y DB_MAP, y se ejecuta la rotación de backups. El resultado informa el tamaño final de la BD (591 MB, 1.19M filas en 70 tablas) y la cuenta de respaldos [3].

Flujo de Ventanas Temporales (Madre + Switch):

```
flowchart LR
  UI([Usuario (Operator UI) abre "Ventana")] -->|API /operator/power/window/
open| Tentaculo
  Tentaculo -->|Reenvía petición| Madre[Servicio Madre]
  Madre --> Validar[Validar lista permitida de servicios]
  Validar --> DockerUp[Ejecutar `docker compose up` en: switch, hermes, ...]
  DockerUp --> WindowReg[Registrar ventana (TTL=300s)]
  WindowReg --> WindowActive[Servicios activos por 300s]
  WindowActive -->|TTL vencido| DockerStop[Ejecutar `docker compose stop`
(stop switch, hermes)]
  DockerStop --> WindowClosed[Ventana cerrada automáticamente]
  WindowClosed -->|Estado| Madre
```

La apertura de una *ventana temporal* implica que la UI invoca al endpoint `/operator/power/window/open`. El gateway (tentáculo) lo reenvía a Madre, que valida la lista de servicios solicitados (e.g. `switch`, `hermes`) y ejecuta `docker compose up` real para arrancarlos [14] [15]. Madre registra la ventana con un TTL (ej. 5 minutos). Al expirar, automáticamente ejecuta `docker compose stop` y cierra la ventana [16]. Mientras dura la ventana, la política es “windowed” y Switch/Hermes están accesibles; fuera de ella, vuelve a `solo_madre`.

Pruebas Derivadas de Diagramas (C)

A partir de los flujos anteriores, proponemos un conjunto de pruebas automáticas y manuales:

- **Smoke tests API**: Usar `pytest` para invocar endpoints clave. Por ejemplo:
- `/operator/api/status` debe responder `{ status:"ok", policy:"solo_madre", ... }` correctamente [17].
- `/operator/api/chat` en modo solo_madre: al enviar `"¿Modo actual?"` se debe obtener código 200, campo `"fallback_source": "local_llm_degraded"` y `degraded: true` [18].
- Verificar rate-limit: hacer 11 solicitudes rápidas de chat con mismo `session_id` y esperar código 429 en la undécima [19].
- Chequear límite de longitud: envío de >4000 caracteres debe retornar 413 Payload Too Large [20].

- **Smoke tests UI**: Con herramientas tipo Selenium/Playwright:

- Cargar `http://<host>:8000/operator/ui/` y confirmar que la página carga sin `NetworkError` (tal como indica paso manual [10]).
- Verificar en DevTools que todas las peticiones fetch sean relativas (`/operator/api/...`) y no contengan `http://localhost:8000` codificado [21].
- En la pestaña Chat de la UI, enviar un mensaje y comprobar en el network que llega `POST /operator/api/chat` y la respuesta tiene los campos esperados (`fallback_source`, `degraded`) [11].

- Verificar en la consola del navegador que hacer `fetch('/operator/api/status')` devuelve JSON apropiado (cód.200 y `policy: "solo_madre"`) [22].

- **Pruebas de fallback**: Alternar servicios:

- Con Switch detenido (solo_madre), el chat debe usar local LLM. Arrancar Switch y confirmar que `fallback_source: "switch_cli_copilot"` (nuevo comportamiento P12) se devuelva.
- Simular fallo de Switch (e.g. contenedor detenido) y confirmar paso al LLM degradado.

- (Opcional) Habilitar variable `VX11_CHAT_ALLOW_DEEPSEEK=1` e interceptar que DeepSeek sea el fallback en ausencia de Switch/LLM [12].

- **Pruebas de retención / post_task**: Ejecutar (simular) `/madre/power/maintenance/post_task` y verificar:

- Los pragmas SQLite se completan sin error [3].
- Se genera un nuevo `DB_MAP_v7_FINAL.md` y `DB_SCHEMA_v7_FINAL.json` (puede comprobarse con scripts existentes).

- Que el backup de BD se creó y que el conteo de respaldos coincide (2 actuales, 23 archivados) [3].

- **Integración en CI**: Automatizar con un pipeline (por ejemplo, GitHub Actions):

- Levantar servicios con `docker-compose -f docker-compose.yml -f docker-compose.override.yml up -d`.
- Ejecutar tests de API con `pytest`.
- Construir frontend (`npm install && npm run build`) y ejecutar pruebas end-to-end (e.g. Playwright o Cypress) para los flujos UI.
- Registrar artefactos de prueba (logs, capturas) en `docs/audit/<TS>/` y reportes de cobertura.

Cada prueba debe incorporar *evidencia trazable* (logs, JSON de respuesta, capturas de pantalla) en la carpeta de auditoría correspondiente, siguiendo el formato de auditoría del proyecto.

Plan de Cierre en 4 Fases (D)

1. **Fase 0 – Verificación Inicial:**
2. Levantar la plataforma en modo `solo_madre`.
3. Correr pruebas de salud: todos los endpoints `/health` deben responder (tentáculo, madre, switch cuando esté arriba).

4. Ejecutar el script de auditoría del flujo completo (`scripts/audit_vx11_flow.py`) para confirmar que registros en BD (routing_events, spawns, pheromone_log, tasks) se generan correctamente [5] [23] .
5. Aplicar las consultas SQLite de integridad (`PRAGMA quick_check; integrity_check; foreign_key_check;`) para asegurarse de que no hay corrupción.
6. Generar y revisar el mapa DB actual (`generate_db_map_from_db`) y compararlo con `DB_SCHEMA_v7_FINAL.json` para detectar cambios inesperados.
7. Documentar el estado actual en un informe preliminar (`docs/audit/<TS>/STEP0_VERIFY.md`) con los hallazgos.

8. **Fase 1 – Corrección UI Chat:**

9. **Cambios de código:** Modificar `operator/frontend/src/services/api.ts` para usar URL relativa:

```
- const BASE_URL = 'http://localhost:8000'
+ const BASE_URL = import.meta.env.VITE_VX11_API_BASE_URL ?? ''
```

   y actualizar `vite.config.ts` tal como se hizo en P12 [9] .
10. **Construcción:** Ejecutar `npm run build` en el frontend y desplegar estático en `tentaculo_link` .
11. **Pruebas:** Realizar smoke tests de UI (pasos del checklist [10]). Verificar que ya **no** aparece `NetworkError` al cargar la UI y que `/operator/api/*` funciona correctamente desde cualquier host.
12. **Commit:** Incluir cambios con mensaje tipo `vx11: Fix Operator UI BASE_URL hardcode` y registrar resultados de pruebas en la carpeta de auditoría.

13. **Fase 2 – Concent. CLI + Fallback LLM:**

14. **Diseño e implementación:** Según la fase D planeada, actualizar la lógica de `/operator/api/chat` en `tentaculo_link/main_v7.py` para que el flujo sea *Switch primero (6s)*, *sino LLM local (2s)*, y solo opcionalmente DeepSeek si `VX11_CHAT_ALLOW_DEEPSEEK=1` . Agregar el campo `fallback_source` en la respuesta con valores `"switch_cli_copilot"` o `"local_llm_degraded"` según corresponda [12] .
15. **Configuración de Switch:** En el módulo Switch, asegurar que existe un proveedor CLI (copilot-cli) que pueda procesar la petición del chat. (Esto se deja para la implementación de fase D/E, pero se debe validar que, si llega un request CLI, el Switch llama al driver correcto).
16. **Pruebas:**
    - Test de **modo switch-only**: Arrancar Switch y enviar chat; debe responder desde Switch con `fallback_source = "switch_cli_copilot"` .
    - Test de **modo solo-madre**: Parar Switch, enviar chat; debe responder con LLM local (`"fallback_source": "local_llm_degraded"` , `"degraded": true`) [18] .
    - Revisar que el tiempo de respuesta entre pasos (switch timeout 6s, LLM 2s) cumple los nuevos SLA.

17. **Commit:**
    `vx11: Implement CLI concentrator + local LLM fallback (Switch-first chat`

`logic)`. Documentar en auditoría los resultados (capturas de logs, JSON de respuesta con `fallback_source`).

18. **Fase 3 – Gates, Evidencia y Post-task:**

19. **Gates (verificación final):** Repasar la lista de comprobación (gates) de P12 [24]:
    - API en solo_madre (chat degradado) pasa.
    - API en modo switch (chat por switch) pasa.
    - URL relativas comprobado.
    - Pruebas de rate-limit y tamaño máximas pasan.
    - Confirmar políticas de seguridad (token en header, sin secretos en frontend).
20. **Evidencia:** Consolidar en `docs/audit/<TS>/` todo el output relevante: logs de `docker compose ps`, resultados JSON de pruebas (`01_api_smoke_tests.json`), capturas de pantalla del UI, lista de auditoría de endpoints, scorecard actualizado, etc.
21. **Post-task:** Ejecutar el endpoint `/madre/power/maintenance/post_task`. Verificar salida: chequear a mano o con script que `quick_check` e `integrity_check` son "ok", y que se generaron los archivos DB_MAP y DB_SCHEMA nuevos [3]. Guardar el informe (JSON o MD) en auditoría.
22. **Commit Final:** Documentar el cierre y hacer `git commit -m "vx11: Phase4 closure – verification, UI fix, CLI fallback, gates & evidence"`, luego hacer push. Incluir instrucciones de fallback: por ejemplo, en caso de falla revertir cambios de UI, arrancar manualmente servicios con `docker-compose up`, etc.

Paquete de Mega-tareas Copilot (E)

A continuación se propone un *workflow* de comandos listos para que Copilot (o un humano) los ejecute fase por fase. Por simplicidad usamos Bash en el directorio raíz del repo (`/home/elkakas314/vx11`):

```
# --- FASE 0: VERIFICACIÓN ---
docker compose up -d
# Esperar que madre y redis estén HEALTHY
python3 -m scripts.audit_vx11_flow
# Ejecutar tests existentes (pytest)
pytest --maxfail=1 --disable-warnings -q
# Revisar integridad DB (script o manual)
sqlite3 data/runtime/vx11.db "PRAGMA quick_check;"
sqlite3 data/runtime/vx11.db "PRAGMA foreign_key_check;"
# Guardar resultados en docs/audit/<TS>/verify_results.txt
echo "RESULTADOS FASE0:" > docs/audit/<TS>/verify_results.txt
docker compose ps >> docs/audit/<TS>/verify_results.txt
sqlite3 data/runtime/vx11.db "SELECT status, value FROM module_status;" >>
docs/audit/<TS>/verify_results.txt
# Commit (solo si todo ok)
git add .
git commit -m "vx11: FASE0 verify - state & DB integrity OK"

# --- FASE 1: FIJO UI ---
# Editar operator/frontend/src/services/api.ts: cambiar BASE_URL a relativo
# (Copilot puede insertar el parche)
```

```
apply_patch << 'EOF'
*** Begin Patch
*** Update File: operator/frontend/src/services/api.ts
@@
-const BASE_URL = 'http://localhost:8000'
+const BASE_URL = import.meta.env.VITE_VX11_API_BASE_URL ?? ''
*** End Patch
EOF
# Editar operator/frontend/vite.config.ts para asegurar proxy /operator/api -
> tentaculo
# (según diseño P12; Omite aquí si ya está)
npm install
npm run build
# Desplegar (copia dist/ a tentaculo_link/ o via compose ya lo monta)
docker compose up -d --no-deps tentaculo_link
# Probar UI en local
# (Opcional: usar script de Puppeteer o similar; aquí sólo comando
placeholder)
echo "Pruebas manual UI..."
# Commit con mensaje claro
git add operator/frontend/src/services/api.ts operator/frontend/
vite.config.ts
git commit -m "vx11: P12 Phase1 - Fix Operator UI base URL to relative"

# --- FASE 2: CLI concentrador + LLM ---
# Modificar el endpoint de chat en tentaculo_link/main_v7.py
apply_patch << 'EOF'
*** Begin Patch
*** Update File: tentaculo_link/main_v7.py
@@ -120,7 +120,15 @@ def operator_chat(request: ChatRequest):
-    # 4) Try Switch (4s) -> 5) DeepSeek (15s) -> 6) degraded
-    resp = try_switch()
-    if not resp: resp = try_deepseek()
+    # 4) **New P12 Flow**: Try Switch (6s timeout)
+    resp = try_switch(timeout=6)
+    if resp:
+        source = "switch_cli_copilot"
+    else:
+        # 5) Fallback: Local LLM (degraded, 2s timeout)
+        resp = try_local_llm(timeout=2)
+        source = "local_llm_degraded"
+        if not resp and os.environ.get("VX11_CHAT_ALLOW_DEEPSEEK", "0") ==
"1":
+            # opcional: DeepSeek si está habilitado
+            resp = try_deepseek()
+            source = "deepseek_api"
+    # 6) Si ninguno dio respuesta, caer en modo degradado general
+    if not resp:
+        resp = "[DEGRADED MODE] Servicio no disponible."
+        source = "degraded"
*** End Patch
```

```
EOF
# Reiniciar tentaculo
docker compose up -d --no-deps tentaculo_link
# Probar chat en ambos modos:
curl -X POST -H "x-vx11-token: vx11-local-token" -H "Content-Type:
application/json" \
    -d '{"message":"test","session_id":"solo"}' http://localhost:8000/
operator/api/chat | tee solo_resp.json
# (Arranca switch y prueba)
docker compose up -d --no-deps switch
curl -X POST -H "x-vx11-token: vx11-local-token" -H "Content-Type:
application/json" \
    -d '{"message":"test","session_id":"switch"}' http://localhost:8000/
operator/api/chat | tee switch_resp.json
# Revisar que 'fallback_source' difiere
grep fallback_source solo_resp.json switch_resp.json
# Commit de cambios
git add tentaculo_link/main_v7.py
git commit -m "vx11: P12 Phase2 - Implement chat flow Switch-first + local
LLM fallback"

# --- FASE 3: GATES, EVIDENCIA, POST_TASK ---
# Ejecutar pruebas finales definidas (API y UI)
pytest --maxfail=1 --disable-warnings -q
# (ejemplo UI: usando Playwright/Cypress, no detallado aquí)
# Ejecutar mantenimiento post-task
curl -X POST -H "x-vx11-token: vx11-local-token" http://localhost:8001/madre/
power/maintenance/post_task | tee post_task_result.json
# Guardar evidencias
sqlite3 data/runtime/vx11.db "SELECT COUNT(*) FROM feats;" >> docs/audit/
<TS>/post_task_log.txt
# Commit final
git add docs/audit/<TS>/post_task_result.json post_task_log.txt
git commit -m "vx11: Phase3 closure - gates passed, evidence collected"
git push origin main
```

En cada paso, si algo falla, Copilot revertirá a la etapa anterior (por ejemplo, deteniendo el fallo, sin avanzar).

Porcentajes y SCORE (F)

Finalmente, se debe recalcular el Score general del sistema usando los artefactos recientes (DB_MAP + SCORECARD). Usando los scripts del repo (`generate_scorecard_complete.py` y similares) se obtendrían métricas como:
- **Orden (orden_fs_pct):** reflejará el % de estructura canónica (por ejemplo, presencia de carpetas esperadas). Se espera cercano a 100% (no se detectaron archivos adicionales y se siguen los invariantes).
- **Estabilidad:** basada en la integridad de la BD (érase `integrity_check = ok`) y ausencia de drift. Dado que los `PRAGMA` han pasado, podemos considerar esta categoría *100%* estable.
- **Routing (coherencia_routing_pct):** medirá consistencia de eventos de routing. Con los tests pasados

(p.ej. `routing_events` presentes [6]) y la tabla canonical de ruteo, estimamos ~100%.
- **Automatización (automatizacion_pct):** basa en la generación de DB_MAP y SCORECARD automáticos, y en un suite de tests semiautomatizada. Con DB_MAP regenerado y SCORECARD actualizado tras post_task [3] , podría rondar 75–100% (dependiendo de si se desarrollaron todos los tests diseño).
- **Autonomía (autonomia_pct):** indica si el sistema se gobierna sólo (e.g. funcionamiento de solo_madre, logs, políticas). Al tener modo solo_madre completo y registro de acciones, se espera alto (por ejemplo 100% si se cumplen todos los chequeos).
- **Global (ponderado):** la media ponderada anterior, probablemente cercana a 100% dada la ausencia de fallos críticos y alta cobertura de tests (0% fallos en la última suite [25]).

Estas cifras deben recalcularse con los scripts oficiales y reportarse en `docs/audit/SCORECARD.json` junto con el desglose (por ejemplo, `"Orden_pct": 100.0`, `"Estabilidad_pct": 100.0`, etc.). Cada porcentaje puede fundamentarse en los conteos de `DB_MAP_v7_META.txt` y resultados de integridad [3] . Se incluirán en el informe final con la justificación basada en el mapeo canónico y el scorecard generado.

Fuentes: Documentación canónica de VX11 [26] [27] , auditoría de interfaz y chat [8] [9] [18] , y resultados del post-task [3] . Cada hallazgo se respalda en estos registros.

---

[1] [2] [4] [27] VX11_STATUS_SNAPSHOT.txt
https://github.com/elkakas314/VX_11/blob/26be729634df447bf43ee666e4e1633c128d5604/VX11_STATUS_SNAPSHOT.txt

[3] [25] VX11_PHASE_1_2_3_4_COMPLETE_FINAL_REPORT.md
https://github.com/elkakas314/VX_11/blob/26be729634df447bf43ee666e4e1633c128d5604/VX11_PHASE_1_2_3_4_COMPLETE_FINAL_REPORT.md

[5] [6] [7] [23] audit_vx11_flow.py
https://github.com/elkakas314/VX_11/blob/26be729634df447bf43ee666e4e1633c128d5604/scripts/audit_vx11_flow.py

[8] P12_OPERATOR_UI_FLOW.md
https://github.com/elkakas314/VX_11/blob/26be729634df447bf43ee666e4e1633c128d5604/docs/audit/P12_OPERATOR_UI_FLOW.md

[9] [12] [24] SUMMARY.md
https://github.com/elkakas314/VX_11/blob/26be729634df447bf43ee666e4e1633c128d5604/docs/audit/20251228T_PROMPT12_P0_DELIVERY/SUMMARY.md

[10] [11] [17] [18] [21] [22] 02_ui_smoke_steps.md
https://github.com/elkakas314/VX_11/blob/26be729634df447bf43ee666e4e1633c128d5604/docs/audit/20251228T_PROMPT12_P0_DELIVERY/02_ui_smoke_steps.md

[13] [19] [20] OPERATOR_CHAT_QUICKREF.md
https://github.com/elkakas314/VX_11/blob/26be729634df447bf43ee666e4e1633c128d5604/OPERATOR_CHAT_QUICKREF.md

[14] [15] [16] routes_power.py
https://github.com/elkakas314/VX_11/blob/26be729634df447bf43ee666e4e1633c128d5604/madre/routes_power.py

[26] VX11_CONTEXT.md
https://github.com/elkakas314/VX_11/blob/26be729634df447bf43ee666e4e1633c128d5604/VX11_CONTEXT.md