OSIRIS V2

Documento de Diseño Definitivo

Arquitectura · Implementación · Hoja de Ruta

Versión 1.0 — Febrero 2026

github.com/osiris-v2/osiris2

1. Visión y Filosofía del Proyecto

1.1 Qué es Osiris

Osiris es una plataforma de computación distribuida modular con capacidades de IA integrada, streaming multimedia, ejecución de código seguro, y comunicación descentralizada entre nodos autónomos.

No es una herramienta puntual. Es una infraestructura sobre la que construir aplicaciones, redes, servicios y lenguajes propios. Su diseño tiene tres propiedades fundamentales que se mantienen en todas las capas:

- Modularidad: cada componente es intercambiable sin romper los demás.
- Escalabilidad: la arquitectura soporta desde un uso local hasta redes distribuidas de nodos.
- Privacidad por diseño: la seguridad no se añade encima, es una propiedad estructural.

1.2 Filosofía de diseño

Cada decisión de arquitectura en Osiris responde a la pregunta: ¿cómo hacemos que esto sea correcto por construcción, no por vigilancia? Esto implica:

- Memoria que se destruye sola al liberarse (URANIO zeroing).
- IA que no puede ejecutar comandos sin pasar por un sandbox validado (CRO).
- Protocolos donde el cliente valida la cabecera antes de aceptar datos (OsirisHeader).
- Lenguajes donde el compilador verifica el espacio de respuestas válidas en tiempo de compilación (FGN pregunta).

1.3 Posición en el ecosistema

Osiris no compite directamente con ningún proyecto existente porque combina capas que normalmente no se encuentran juntas. El punto más cercano de comparación sería una combinación de Lua (VM embebible), Ethereum (contratos ejecutables con consenso), y un servidor de streaming — pero con orientación a privacidad, IA local, y una sintaxis de lenguaje propia.

2. Mapa Completo de Componentes

2.1 Visión de capas

El sistema se organiza en cinco capas. Cada capa tiene componentes con estado de implementación definido.

CAPA	NOMBRE	COMPONENTES
5 — Distribución	Plataforma base	Installer .deb, Git sparse-checkout, bio.deb, política release/dev
4 — Plataforma	Runtime Python	com.py, ops.c, CRO parser, Apache/Nginx/PHP/MariaDB/Docker
3 — Comunicación	Red y IA	Protocolo Osiris, server.rs, streamAI.py, gemini2.py, API Translator
2 — Runtime	C/Rust Engine	VM + computed gotos, RB_SafePtr/Uranio, ODS Engine, Cerebro/Nodo
1 — Lenguaje	FGN/Bytecode	Compilador FGN, FGN_Opcode96, red distribuida, blockchain (proyectado)

2.2 Estado actual de implementación

Componente	Estado	Prioridad	Notas
Installer .deb / Git	**COMPLETO**	—	Release y dev diferenciados, sparse-checkout
bio.deb escritorio	**COMPLETO**	—	Fase desarrollo, aplicaciones gráficas
com.py — kernel plugins	**COMPLETO**	—	Carga dinámica, historial, señales
ops.c — shell macros	**COMPLETO**	—	Variables persistentes, fork/execvp
CRO parser/translator	**COMPLETO**	—	Sandbox IA, DevMode/Producción
gemini2.py	**COMPLETO**	—	Sesiones, TTS, SFTP, Tkinter, cifrado
streamAI.py	**COMPLETO**	—	Cliente WebSocket universal con reconexión
fftv.py — streaming	**COMPLETO**	—	Perfiles YT/Rumble/HLS, multiprocess
server.rs — WS+Ollama	**COMPLETO**	—	Sesiones, historial, comandos
OsirisHeader C/Rust	**BLOQUEANTE**	CRÍTICA	Desincronización C vs Rust — fix listo
RB_SafePtr unificado	**BLOQUEANTE**	CRÍTICA	4 definiciones conflictivas — fix listo
#include .c en main()	**PENDIENTE**	Alta	fgn_math.c incluido dentro de main()
payload_size validación	**PENDIENTE**	Alta	Sin límite superior — riesgo 4GB alloc
XOR dinámico por sesión	**PENDIENTE**	Alta	Stream viaja en claro actualmente

API Translator	**PENDIENTE**	Alta	Ollama/Gemini/OpenAI — arquitectura lista
HMAC-SHA256 signer	**PENDIENTE**	Media	Firma actual es solo 32-bit hash fijo
TLS canal TCP	**PENDIENTE**	Media	Requiere tokio-rustls
Renderer SDL2 propio	**PENDIENTE**	Media	Reemplaza ffplay, habilita privacidad total
Gramática formal FGN	**PENDIENTE**	Media	Especificación antes del compilador
Compilador FGN	**PROYECTADO**	Fase 3	Lexer + parser + emisor de bytecode
VM FGN red distribuida	**PROYECTADO**	Fase 4	Nodos autónomos, descubrimiento DHT
Blockchain consenso	**PROYECTADO**	Fase 4	Estado compartido entre nodos

3. Decisiones de Diseño Fijadas

Estas decisiones NO se reabren. Están respaldadas por código existente y cualquier cambio requeriría reescritura de múltiples componentes. Se documentan aquí para que sirvan de ancla.

3.1 Protocolo de red — OsirisPacket

Decisión fijada: cabecera de 16 bytes con layout exacto

Byte 0	version (valor 2)
Byte 1	seed_id — identificador del nodo emisor
Byte 2	opcode — tipo de operación
Bytes 3-6	signature — firma u32 little-endian
Bytes 7-10	payload_size — tamaño u32 little-endian
Bytes 11-15	padding — reservado, debe ser cero

El struct C OsirisHeader debe verificarse en compilación con:

```
_Static_assert(sizeof(OsirisHeader) == 16, "Header desincronizado");
```

> **El bug actual es que main.c leía magic(12)+chunk_size(4) en lugar de este layout. El fix está en main_header_fix.c entregado.**

Opcodes fijados

5 — RESCALE	Redimensionar buffer Uranio del receptor
7 — VIDEO	Chunk de video MPEG-TS sigue al header
9 — EXIT	Cerrar conexión limpiamente
10 — PAUSE	Alternar pausa de reproducción
15 — SKIP	Saltar sin cerrar el pipe
22 — IA_UPDATE	Inyectar bloque ADN de modelo IA

Canal dual — puertos 2000 y 2001

Puerto 2000: canal DATA (video, alto volumen). Puerto 2001: canal CONTROL (comandos, baja latencia). Esta separación garantiza que los comandos nunca bloqueen detrás de chunks de video grandes. Es invariante de arquitectura.

3.2 Sistema de memoria — RB_SafePtr

Decisión fijada: un único RB_SafePtr en rb_csp.h

Existían cuatro definiciones conflictivas. La decisión es: rb_csp.h es la única fuente de verdad. Las demás se eliminan o renombran.

```
typedef struct {     void*      data;       // puntero real al heap     uint32_t
size;       // capacidad en bytes     Hardness   hardness;   // ACERO / DIAMANTE /
URANIO     uint32_t*  ref_count;  // contador compartido     FaseEstado
```

```
estado;     // PARTICULA / BIFURCADO / VOID     double     coherencia; // integridad [0.0-1.0] } RB_SafePtr;
```

Niveles de dureza — invariantes

- ACERO: memoria temporal, sin protecciones especiales, liberación normal.
- DIAMANTE: estructuras de larga vida (compilador, handlers), sin zeroing garantizado.
- URANIO: memoria crítica — rb_liberar() ejecuta memset(data, 0, size) ANTES de free(). Invariante absoluto.

> **ODS_SafeRef (antes llamado RB_SafePtr en ods_definiciones.h) es un descriptor de auditoría, NO un allocator. Su rename es parte del fix pendiente.**

Bifurcación/Colapso — zero-copy sharing

rb_bifurcar_onda(): dos punteros al mismo bloque físico, ref_count++, coherencia=0.5 cada uno. rb_colapsar_observacion(): merge a propietario único, ref_count--, coherencia=1.0. Caso de uso principal: mismo frame de video leído por renderer y analizador FGN simultáneamente sin copia.

3.3 Sistema CRO — sandbox de ejecución IA

Decisión fijada: toda acción de IA pasa por CROParser

La IA no ejecuta comandos directamente. Genera bloques CRO que el parser valida contra cro_definitions, el translator convierte a comandos reales, y el sistema decide confirmación según el modo.

```
```CRO FILE_* READ_FILE PATH="/var/osiris2/bin/config.json" ```
```

- Parámetros DYNAMIC: la IA puede rellenarlos.
- Parámetros estáticos: la IA no puede cambiarlos aunque lo intente.
- needs_confirmation=True en DevMode: el operador humano aprueba antes de ejecutar.

> **La clave de Fernet para cifrar API keys está hardcodeada en gemini2.py. En Fase 2 debe moverse a variable de entorno o archivo de configuración seguro fuera del repo.**

3.4 Tres shells distintas — no confundir

El proyecto tiene tres intérpretes de comandos en capas distintas:

ops.c	Shell de macros bash. Variables = comandos shell. Persistencia en .vars. Ejecuta con fork/execvp. Es la ODS de la plataforma Python base.
ods_ejecutor.c	Engine ODS del runtime C/Rust. Operadores: ~, #, $, @, #!. Transmite al Nodo vía TCP. Auditoría por hash djb2. Variables con firma de integridad.
FGN (proyectado)	Lenguaje compilado a bytecode. Variables = handles en h_table. Bytecode FGN_Opcode96. Ejecuta en VM distribuida entre nodos remotos.

ops.c puede cargarse sobre ods_ejecutor.c como capa de ejecución real del sistema operativo. ods_ejecutor.c actúa como capa de autorización encima de ops.c. FGN compilará a llamadas que eventualmente llegan a ops.c para ejecución real.
```

3.5 com.py — contrato del sistema de plugins

Todo módulo cargable por com.py debe cumplir:

- Existir como archivo .py en el directorio bin/com/
- Tener una función main(args) con exactamente un argumento.
- El argumento args es una lista de strings (equivalente a argv).

com.py verifica la firma con AST antes de importar — si la función no existe o tiene firma incorrecta, el módulo no se carga. Este contrato no cambia. Los módulos nuevos deben respetarlo.

4. Contratos Entre Capas

Los contratos definen cómo se comunican los componentes. Un contrato roto es un bug de integración. Documentarlos aquí permite trabajar en una capa sin romper las demás.

4.1 com.py ↔ módulos de comandos

- com.py llama: module.main(args: list[str])
- El módulo devuelve: cualquier valor (ignorado por com.py) o None.
- El módulo puede imprimir a stdout libremente.
- El módulo NO debe terminar el proceso con sys.exit() — usar return.
- Señales: com.py gestiona SIGINT. El módulo puede capturar sus propias señales pero debe restaurarlas.

4.2 CROParser ↔ IA (Gemini/Ollama)

- La IA genera bloques delimitados por ```CRO ... ``` en su respuesta de texto.
- Formato de iniciador: GRUPO_* MIEMBRO1, MIEMBRO2
- Formato de parámetro: NOMBRE="valor"
- Formato multilínea: NOMBRE=<<<DELIMITADOR ... DELIMITADOR
- Formato texto largo: NOMBRE=""" ... """
- La IA solo puede usar grupos y miembros definidos en cro_definitions.
- El parser ignora silenciosamente parámetros no definidos como DYNAMIC.

4.3 Cerebro (Rust) ↔ Nodo (C)

- Cerebro envía siempre primero el OsirisHeader de 16 bytes.
- Si payload_size > 0, los bytes del payload siguen inmediatamente al header.
- El Nodo valida: version==2, payload_size <= URANIO_MAX_BLOQUE (64MB).
- Si la validación falla: el Nodo cierra la conexión, no intenta recuperarse.
- Canal DATA (2000): solo opcodes 5, 7, 9, 15.
- Canal CONTROL (2001): solo opcodes 9, 10, 22.

4.4 server.rs ↔ streamAI.py (WebSocket)

- El cliente envía: texto plano UTF-8 (el mensaje del usuario).
- El servidor responde: fragmentos de texto en streaming, uno por mensaje WebSocket.
- Fin de respuesta: el servidor deja de enviar. El cliente detecta el fin por timeout de 2 segundos (configurable).
- El servidor mantiene historial de sesión en memoria — NO en disco.
- Al desconectar: el servidor destruye la sesión completa incluyendo historial.

4.5 API Translator ↔ backends IA

Contrato de la interfaz unificada que todos los backends deben implementar:

```
class AIBackend:     async def stream(prompt, context, model) -> AsyncIterator[str]
async def list_models() -> list[str]     async def health() -> bool     def
name(self) -> str
```

- stream() debe ser un generador async que yield fragmentos de texto.
- list_models() devuelve lista de IDs de modelos disponibles para ese backend.
- health() devuelve True si el backend responde, False si no.
- El proxy expone la misma interfaz WebSocket que server.rs — streamAI.py no cambia.

5. Arquitectura de Privacidad

La privacidad en Osiris no es una característica añadida — es una consecuencia de las decisiones de arquitectura. Se documenta aquí el modelo de amenaza real y qué protege cada capa.

5.1 Capas de protección

Capa 1 — Zero Persistence (IMPLEMENTADA)

✓ **Nada del sistema escribe contenido sensible en disco durante operación normal.**
✓ **rb_liberar() ejecuta memset(data, 0, size) antes de free() en todos los bloques URANIO.**
✓ **Al desconectar, la sesión completa se destruye atómicamente.**
✓ **Los mensajes de texto del usuario nunca se escriben en /tmp, logs, ni bases de datos locales.**

Capa 2 — Cifrado de contenido en tránsito (PENDIENTE)

◎ XOR dinámico por frame: clave de sesión única negociada en handshake, deriva por frame_id.
◎ TLS sobre TCP: protege el canal completo incluyendo headers del protocolo.

Con ambas capas activas, un analizador de red no puede determinar el tipo de operación ni el contenido.

Capa 3 — Texto como píxeles (REQUIERE RENDERER PROPIO)

El Cerebro renderiza mensajes de texto como imagen pequeña. Los empaqueta como chunk de video opcode 7. El Nodo los recibe como datos de video — nunca como strings. Un volcado de RAM del Nodo buscando texto legible no encontraría nada.

> **Esto requiere el renderer SDL2/OpenGL propio. Con ffplay actual, los frames pasan por un proceso externo sin las protecciones de memoria de Osiris.**

Capa 4 — Opacidad ante el SO (PARCIAL)

✓ **Con renderer propio: el SO ve un único proceso dibujando píxeles. No hay proceso multimedia externo.**
✓ **RB_SafePtr garantiza que no haya desbordamientos — los EDR usan desbordamientos como señal de detección.**
◎ Anti-debug: prctl(PR_SET_DUMPABLE, 0) para deshabilitar core dumps.
◎ mlock() para buffers URANIO críticos — previene swap a disco.

5.2 Lo que Osiris NO garantiza

> **No protege contra el kernel comprometido. Si el SO está bajo control de un atacante, ninguna medida en espacio de usuario es suficiente.**
>
> **No oculta la existencia de conexiones TCP. Un firewall ve el tráfico entre IPs en los puertos configurados.**
>
> **El XOR por frame no es cifrado criptográfico fuerte — proporciona opacidad de contenido pero no autenticación. TLS + HMAC son necesarios para garantías criptográficas reales.**

6. Lenguaje FGN — Especificación Preliminar

FGN es el lenguaje de programación distribuida de Osiris. Un programa FGN es simultáneamente el protocolo de red, la definición de UI, la lógica de negocio, y el contrato de interacción. Se compila a bytecode FGN_Opcode96 y se ejecuta en la VM de los nodos.

6.1 Filosofía del lenguaje

- Un programa FGN es un contrato de interacción. El receptor ve las reglas antes de ejecutar.
- El mismo bytecode corre en cualquier nodo — local, remoto, o distribuido.
- El programa define la UI, la lógica, y el canal de comunicación simultáneamente.
- Las respuestas válidas están declaradas en el bytecode — el compilador verifica cobertura completa.

6.2 Modos de operación

Standalone local	Sin conexión. El nodo ejecuta bytecode propio. Aplicaciones locales, interfaces gráficas, procesamiento de datos.
Invitación	Un nodo envía bytecode a otro. El receptor decide ejecutar. Las reglas están declaradas en el programa.
Red dinámica	Múltiples nodos conectados. Estado compartido vía blockchain. Consenso distribuido. Sin nodo central.

6.3 Sintaxis — elementos fijados

Cabecera de conexión

```
conecta 192.168.1.10:2000 conecta ia-server.local:8081
```

Siempre va primero. Puede haber múltiples conexiones simultáneas. Sin conecta, el programa corre en modo standalone.

Declaración de recursos

```
video 1 /var/videos/x.mp4 ventana 1 800x600 "Mi App" audio 1 /var/audio/intro.mp3
```

El número es el handle. El compilador lo registra en la h_table. El runtime lo vincula al recurso.

Envío de texto multilínea

```
enviaTexto<<<   Hola, ¿quieres ver este vídeo?   1:title   1:descripcion />>>
```

1:title y 1:descripcion son interpolaciones del handle 1. El compilador las resuelve contra la h_table.

Bloques de decisión tipados

```
pregunta si/no/quizas :   si      -> 1:play   no      -> msgCierra "Vale, adios"
quizas  -> >>>                chat conecta              esperaRespuesta
si acuerdo -> 1:play                sino -> msgCierra "Quizas otro
dia"            />>>   timeout -> msgCierra "Sin respuesta"   sino    -> ERROR
/pregunta
```

Las ramas están definidas por el tipo de respuesta esperada declarado en la firma del bloque (si/no/quizas). El compilador verifica que todas las ramas posibles estén cubiertas o exista sino como catch-all.

Canal bidireccional >>>

```
quizas -> >>>     chat conecta     esperaRespuesta     si acuerdo -> 1:play     sino
-> msgCierra "Quizas otro dia" />>>
```

>>> abre un canal de comunicación bidireccional en tiempo real. La ejecución determinista se pausa. Cuando se cumple la condición de salida, el flujo del programa se reanuda. Es la apertura de sesión interactiva dentro del flujo del programa.

Consenso distribuido

```
consenso 3/5 :   nodo cerebro-a   nodo cerebro-b   nodo cerebro-c   nodo cerebro-d
nodo cerebro-e /consenso  si consenso.ok -> ejecutar bloque-critico sino -> ERROR
"Sin consenso de red"
```

6.4 Compilación a FGN_Opcode96

conecta ip:puerto	OP_SONDA_HW + payload de dirección
video 1 /path	OP_MIRROR_INI | id=1 | payload=hash_de_ruta
enviaTexto<<<...>>>	OP_TEXT_INJECT | id=0 | payload=longitud del bloque
pregunta si/no/quizas	OP_SYNC_URANIO + tabla de saltos a las ramas
1:play	OP_MIRROR_INI | id=1 | payload=0 (reproducir)
msgCierra "texto"	OP_TEXT_INJECT + OP_MIRROR_OFF | id=1 | payload=0
>>>	OP_BIFURCAR_ONDA — pausa determinismo, abre canal bidireccional
consenso N/M	OP_COLAPSAR — espera confirmación de N de M nodos

6.5 Relación CRO / FGN / ODS

Son tres capas de lenguaje distintas en roles distintos:

CRO	Lenguaje IA->sistema. Vertical: controla la relación IA-SO en una máquina. No es visible para el usuario final.
ODS (ops.c)	Shell de macros bash. Horizontal: macros de comandos del sistema operativo para el operador humano.
FGN	Lenguaje de programación distribuida. Horizontal: coordina ejecución entre múltiples nodos en red. Es el lenguaje del usuario/desarrollador de Osiris.

Una IA podría generar CRO que compile y envíe bytecode FGN a un nodo remoto, que a su vez llame a ops.c para ejecutar un comando bash. Esa es la cadena de capas completa.

7. API Translator — Diseño

El API Translator es una capa de abstracción que expone una interfaz WebSocket unificada hacia los clientes, y traduce hacia la API específica de cada backend de IA. Los clientes (streamAI.py, gemini2.py, ODS, FGN) no necesitan saber con qué modelo están hablando.

7.1 Arquitectura

```
CLIENTES streamAI.py / gemini2.py / ODS / FGN            |     [ WebSocket
unificado ]            |     [ AIProxy — aiproxy.py ]     /        |         \ Ollama
Gemini     OpenAI (local)  (remota)   (remota)
```

7.2 Interfaz de backend

```
class AIBackend:     async def stream(self, prompt, context, model) ->
AsyncIterator[str]     async def list_models(self) -> list[str]     async def
health(self) -> bool     def name(self) -> str
```

7.3 Implementaciones

OllamaBackend	HTTP a localhost:11434. Modelo configurable. Base ya en server.rs — migrar a Python.
GeminiBackend	SDK google.generativeai. Lógica ya en gemini2.py — extraer y adaptar.
OpenAIBackend	API REST estándar. Compatible con cualquier proveedor que implemente el estándar OpenAI.
AnthropicBackend	API REST Anthropic. Misma estructura, distinto endpoint y formato.

7.4 Lógica del proxy

- Enrutamiento: por defecto usa el backend activo configurado.
- Fallback: si el backend activo falla health(), pasa al siguiente en la lista de prioridad.
- Cambio de backend: via CRO con un nuevo grupo AI_* o via argumento en streamAI.py.
- El cliente no percibe el cambio de backend — la interfaz WebSocket es idéntica.

7.5 Integración con com.py

aiproxy.py se instala como comando de com.py: mount aiproxy desde la consola Osiris. La función main(args) acepta: start, stop, status, backend [nombre], models.

8. Hoja de Ruta

FASE 2A — Estabilización crítica

Objetivo: cerrar los bugs bloqueantes. Estimación: 1-2 semanas. Sin esto, todo lo que se construya encima es sobre arena.

Tarea 1 — Fix OsirisHeader (CRÍTICA)

✓ **Código generado: main_header_fix.c y protocol.rs actualizados.**

Aplicar el OsirisHeader struct en main.c con _Static_assert. Verificar con wireshark que los opcodes se reciben correctamente.

Criterio de completitud: el Nodo recibe opcode 7 para video, opcode 9 para EXIT, sin confusión.

Tarea 2 — Unificar RB_SafePtr (CRÍTICA)

✓ **Código generado: rb_csp.h maestro y ods_definiciones.h con ODS_SafeRef.**

Eliminar las definiciones de uranio.h y fgn_ai_core.h. Actualizar todos los include. Compilar y verificar que no hay redefiniciones.

Criterio de completitud: proyecto compila sin warnings de redefinición de tipos.

Tarea 3 — Fix #include .c en main()

Mover FGN_Forjar de demo/fgn_math.c a un archivo .h con declaración y .c con definición. Incluir el .h en main.c. Llamar a FGN_Forjar normalmente.

Criterio de completitud: main() no tiene ningún #include dentro del cuerpo de la función.

Tarea 4 — Validar payload_size

En la función que procesa opcode RESCALE (5), añadir antes de rb_rescale:

```
if (header.payload_size > URANIO_MAX_BLOQUE) {     fprintf(stderr, "payload_size excede limite\n");     close(fd); return; }
```

Criterio de completitud: enviar un paquete con payload_size=0xFFFFFFFF no crashea el Nodo.

FASE 2B — Privacidad del canal

Objetivo: hacer que el stream sea técnicamente privado. Estimación: 2-3 semanas.

Tarea 5 — Handshake de clave de sesión

Al conectar, el Cerebro genera 32 bytes aleatorios (rand_bytes en Rust), los envía cifrados con una clave pública del Nodo (o por ahora con Diffie-Hellman simple). El Nodo almacena la clave en un bloque URANIO que se zeroa al desconectar.

Tarea 6 — XOR dinámico por frame

Cerebro: antes de enviar cada chunk, aplicar XOR con derive_frame_key(session_key, frame_id). Nodo: después de leer del buffer Uranio, antes de pasar al renderer, aplicar la misma operación inversa.

```
fn xor_frame(data: &mut [u8], session_key: &[u8], frame_id: u64) {     let key =
derive_frame_key(session_key, frame_id);     for (b, k) in
data.iter_mut().zip(key.iter().cycle()) {         *b ^= k;     } }
```

Tarea 7 — HMAC-SHA256 en signer.rs

Reemplazar el hash de 32 bits con semilla fija por HMAC-SHA256 usando la clave de sesión. Esto convierte la firma de detección de corrupción accidental a autenticación real del origen.

FASE 2C — API Translator

Objetivo: abstracción de backends de IA. Estimación: 1-2 semanas.

1. Crear aiproxy.py con la clase AIBackend abstracta.
2. Implementar OllamaBackend (migrar lógica de server.rs).
3. Implementar GeminiBackend (extraer lógica de gemini2.py).
4. Implementar el proxy WebSocket con enrutamiento y fallback.
5. Registrar como comando com.py: mount aiproxy.
6. Actualizar streamAI.py con parámetro --backend.

FASE 3 — Lenguaje FGN

Objetivo: compilador funcional básico. Estimación: 1-2 meses.

7. Gramática formal BNF del lenguaje FGN (1 semana).
8. Lexer: tokenizador de la sintaxis FGN (1 semana).
9. Parser: árbol de sintaxis abstracta (AST) (2 semanas).
10. Emisor de bytecode: AST -> FGN_Opcode96 (2 semanas).
11. Extensión de la VM para los nuevos opcodes: conecta, ventana, pregunta, >>>, consenso.
12. Renderer SDL2 propio — eliminar dependencia de ffplay.

FASE 4 — Red distribuida

Objetivo: nodos autónomos que se descubren y comunican. Estimación: 2-4 meses.

13. Descubrimiento de nodos: mDNS para red local, DHT para internet.
14. Routing de bytecode FGN entre nodos.
15. Estado compartido: blockchain ligero para consenso.
16. Protocolo de consenso N/M implementado.
17. Identidad de nodo: clave pública como identidad permanente.

FASE 5 — Distribución Debian propia

Objetivo: repositorio APT propio y meta-paquete. Estimación: futuro.

- Repositorio APT en dominio propio (apt.osiris.xxx).
- Meta-paquete osiris2 que gestiona todas las dependencias.
- La infraestructura gitup-release.txt + sparse-checkout ya es la base de esto.
- bio.deb como paquete de escritorio oficial.

9. Tareas Pendientes Inmediatas — Checklist

Código ya generado en sesión de trabajo. Listo para aplicar y probar.

9.1 Código listo para aplicar

Fix OsirisHeader

✓ **Archivo: /mnt/user-data/outputs/main_header_fix.c**

Contiene: OsirisHeader struct, _Static_assert(16), lectura correcta del header, validación de version==2 en lugar de magic. Reemplaza la sección de lectura de header en main.c.

RB_SafePtr unificado

✓ **Archivo: /mnt/user-data/outputs/rb_csp.h**

Contiene: definición única maestra de RB_SafePtr con todos los campos correctos, Hardness enum, FaseEstado enum, declaraciones de funciones.

ODS_SafeRef renombrado

✓ **Archivo: /mnt/user-data/outputs/ods_definiciones.h**

Contiene: ODS_SafeRef (antes RB_SafePtr en ods_definiciones.h), sin conflicto con rb_csp.h. Los archivos ods_memoria.c y ods_ejecutor.c deben actualizar el campo nombre a safe_ref.

Protocol.rs sincronizado

✓ **Archivo: /mnt/user-data/outputs/protocol.rs**

Contiene: OsirisPacket struct en Rust sincronizado byte a byte con OsirisHeader en C. Serialización explícita en lugar de transmute.

9.2 Pendiente de escribir

◎ Handshake de clave de sesión (Fase 2B, Tarea 5).
◎ XOR dinámico por frame en Cerebro y Nodo (Fase 2B, Tarea 6).
◎ HMAC-SHA256 en signer.rs (Fase 2B, Tarea 7).
◎ aiproxy.py con backends OllamaBackend y GeminiBackend (Fase 2C).
◎ Gramática BNF formal de FGN (inicio de Fase 3).
◎ Mover clave Fernet de gemini2.py a variable de entorno.

9.3 Pruebas de regresión recomendadas

18. Tras aplicar OsirisHeader: enviar stream desde Cerebro, verificar que el Nodo reproduce video y responde correctamente a opcode EXIT.
19. Tras unificar RB_SafePtr: compilar proyecto completo con -Wall -Wextra, verificar cero warnings de redefinición.
20. Tras fix payload_size: enviar paquete malformado con payload_size=0xFFFFFFFF, verificar que el Nodo cierra conexión sin crash.
21. Tras XOR por frame: capturar tráfico con tcpdump, verificar que los bytes no son MPEG-TS reconocible.

10. Glosario

Cerebro Semilla	Servidor Rust. Transcode video con FFmpeg, transmite vía TCP con protocolo Osiris.
Nodo Músculo	Cliente C. Recibe stream, renderiza (ffplay/SDL2), ejecuta bytecode remoto.
OsirisHeader	Cabecera de 16 bytes del protocolo de red. Layout: version(1)+seed_id(1)+opcode(1)+signature(4)+payload_size(4)+padding(5).
RB_SafePtr	Smart pointer de Osiris con ref-counting, niveles de dureza, y estado de bifurcación.
URANIO	Nivel de dureza máximo. Garantiza zeroing de memoria antes de liberación.
DIAMANTE	Nivel de dureza para estructuras de larga vida. Sin zeroing garantizado.
ACERO	Nivel de dureza para memoria temporal. Sin protecciones especiales.
ODS_SafeRef	Descriptor de auditoría ligero del ODS Engine. NO es un allocator.
Bifurcación	Dos RB_SafePtr apuntando al mismo bloque físico (zero-copy sharing).
Colapso	Reunificación de dos SafePtr bifurcados en un único propietario.
CRO	Command Request Object. Lenguaje intermedio IA->sistema. Sandbox de ejecución segura.
FGN	Lenguaje de programación distribuida de Osiris. Se compila a FGN_Opcode96.
FGN_Opcode96	Formato de bytecode de la VM de Osiris. 96 bits por instrucción.
h_table	Tabla de handles del runtime. Vincula IDs numéricos a recursos (video, ventana, audio).
ops.c	Shell de macros bash. ODS de la capa Python base de Osiris.
ods_ejecutor.c	Engine ODS del runtime C/Rust. Operadores: ~, #, $, @, #!.
Canal DATA	Puerto 2000. Tráfico de video, alto volumen.
Canal CONTROL	Puerto 2001. Comandos, baja latencia.
API Translator	Proxy WebSocket que abstrae múltiples backends de IA bajo una interfaz unificada.
DevMode	Modo CRO donde cada acción propuesta por la IA requiere confirmación del operador humano.
HMAC	Hash-based Message Authentication Code. Autenticación criptográfica de paquetes.
DHT	Distributed Hash Table. Protocolo de descubrimiento de nodos en redes P2P.

sparse-checkout	Técnica Git para descargar solo los archivos listados en gitup-release.txt.