GameStringer - Technical Deep Dive

🎮 Cos'è GameStringer?

GameStringer è un **software desktop professionale** per la localizzazione di videogiochi con **due modalità operative**:

1. **Offline Translation** - Tool per localizzatori professionisti
2. **Real-Time Injection** - Traduzione runtime durante il gameplay

---

🧬 Stack Tecnologico

Frontend

- **Framework**: Next.js + React
- **Container**: Tauri (NON Electron!)
- **Styling**: TailwindCSS + shadcn/ui
- **Icons**: Lucide React
- **State Management**: React Hooks + Context

Backend

- **Core**: Rust (via Tauri)
- **System APIs**: WinAPI per memory manipulation
- **Database**: SQLite locale
- **Threading**: Tokio async runtime + std::thread
- **Serialization**: Serde (JSON)

Perché Tauri e non Electron?

- **Performance**: Rust backend nativo vs Node.js
- **Dimensioni**: ~10MB vs ~150MB
- **Sicurezza**: Memory safety di Rust
- **Accesso sistema**: WinAPI diretta senza overhead

📚 Modalità 1: Offline Translation

Tool professionale per localizzatori che lavorano su progetti di gioco.

Funzionalità

- **Import/Export** multi-formato:
    - JSON, PO (gettext), RESX, CSV, XLIFF, XML
- **Translation Memory** (TM) locale
    - Gestita in Rust per performance
    - Lookup velocissimo su milioni di entry
- **Quality Gates**
    - Validazione automatica traduzioni
    - Controllo coerenza terminologica
    - Rilevamento placeholder mancanti
- **Content Classifier**
    - Classifica contenuti per contesto AI
    - UI, Dialog, Tutorial, Lore, ecc.
- **Batch Translation**
    - Integrazione LLM (Claude, OpenAI, DeepL)
    - Traduzione parallela con rate limiting
    - Gestione errori e retry automatici

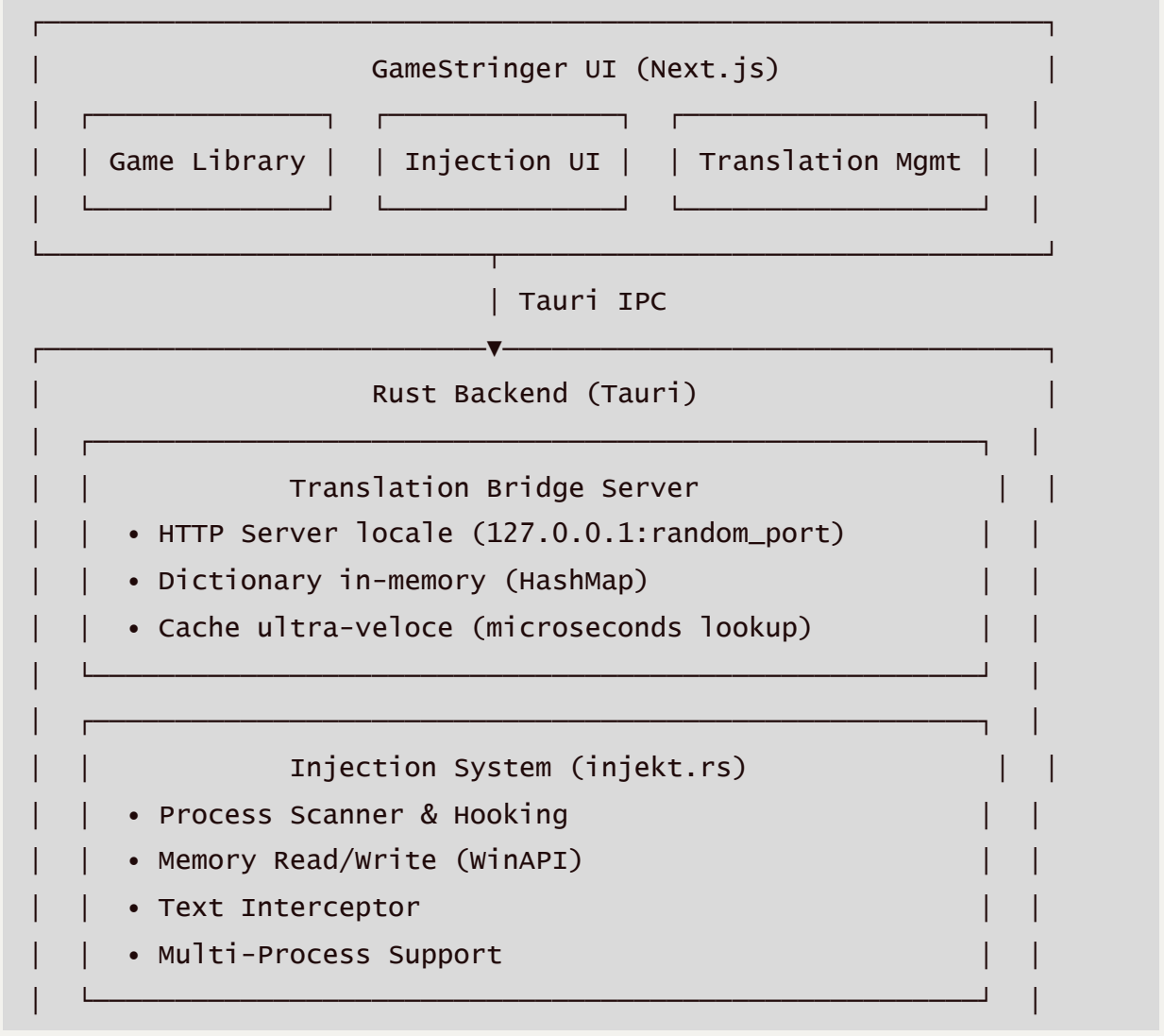
Workflow Tipico

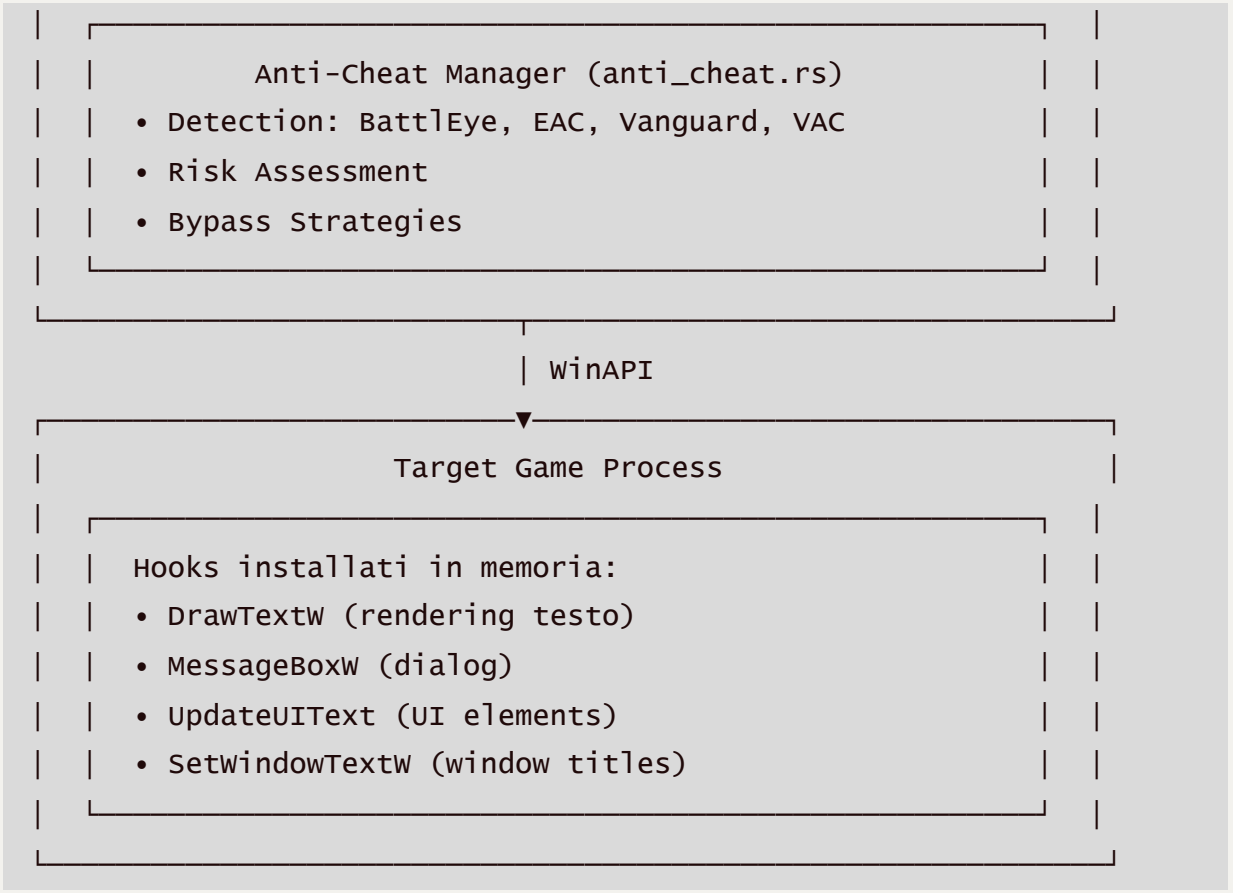
```
1. Importa file localizzazione (es. en.json)
2. Carica Translation Memory esistente
3. Pre-traduci con TM (match esatti)
4. Traduci rimanenti con AI (batch)
5. Applica Quality Gates
6. Revisione manuale
7. Esporta file tradotto (es. it.json)
```

🎯 Modalità 2: Real-Time Injection

Questa è la parte più interessante - traduzione runtime durante il gameplay!

Architettura Generale





🔍 Come Funziona l'Injection

1. Game Library Scanning

Implementato in: `src-tauri/src/commands/steam.rs`, `epic.rs`, `gog.rs`, ecc.

```
// Scan parallelizzato di tutte le piattaforme
async fn get_games_fast() {
    let handles = vec![
        tokio::spawn(scan_steam()),
        tokio::spawn(scan_epic()),
        tokio::spawn(scan_gog()),
        tokio::spawn(scan_origin()),
        tokio::spawn(scan_ubisoft()),
        tokio::spawn(scan_battlenet()),
        tokio::spawn(scan_itch()),
        tokio::spawn(scan_rockstar()),
    ];
```

```
    // Attendi tutti i risultati
    let results = join_all(handles).await;
}
```

Parsing .vdf di Steam: ✅ Implementato

- Legge `libraryfolders.vdf` per trovare percorsi librerie
- Parsa `appmanifest_*.acf` per info giochi
- Estrae: nome, ID, dimensione, ultimo avvio, ecc.

2. Process Detection & Attachment

File: `src-tauri/src/injekt.rs` (1008 righe)

```
// Trova processo target
pub async fn find_processes() -> Vec<ProcessInfo> {
    unsafe {
        let snapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS,
0);
        // Enumera tutti i processi
        // Filtra solo giochi (euristica basata su nome/path)
        // Ritorna lista processi
    }
}

// Apri handle al processo
let process_handle = OpenProcess(
    PROCESS_ALL_ACCESS,
    FALSE,
    target_pid
);
```

3. Anti-Cheat Detection

File: `src-tauri/src/anti_cheat.rs` (429 righe)

Prima di iniettare, il sistema rileva anti-cheat attivi:

```
pub struct AntiCheatManager {
    known_systems: HashMap<String, AntiCheatInfo>,
    // BattlEye, EAC, Vanguard, VAC, Denuvo, ecc.
}

pub enum RiskLevel {
    Low,      // Injection sicura
    Medium,   // Richiede precauzioni
    High,     // Molto rischioso
    Critical, // Sconsigliato (ban risk)
}

pub enum BypassStrategy {
    DelayedInjection,      // Aspetta finestra sicura
    ProcessHollowing,      // Tecnica avanzata
    ManualMapping,         // Evita LoadLibrary
    ReflectiveDLL,         // DLL in memoria
    ThreadHijacking,       // Hijack thread esistente
    SetWindowsHook,        // Hook Windows standard
    WaitForSafeWindow,     // Aspetta momento sicuro
}
```

Esempio: Se rileva BattlEye → Risk: High → Strategy: Delayed + WaitForSafeWindow

4. Hook Installation

Come trova i punti di aggancio?

```
// Trova moduli caricati nel processo
unsafe {
    let snapshot = CreateToolhelp32Snapshot(TH32CS_SNAPMODULE,
pid);
    Module32First(snapshot, &mut module_entry);
```

```
    // Per ogni modulo (game.exe, user32.dll, ecc.)
    while Module32Next(snapshot, &mut module_entry) {
        // Cerca funzioni comuni di rendering testo
        let functions = [
            "DrawTextW",
            "DrawTextExW",
            "TextOutW",
            "MessageBoxW",
            "SetWindowTextW",
            // + pattern matching per funzioni custom del gioco
        ];

        // Installa hook su ogni funzione trovata
        install_hook(module_base + function_offset);
    }
}
```

Hook Point Structure:

```
struct HookPoint {
    address: usize,              // 0x00401000
    original_bytes: Vec<u8>,     // Backup per ripristino
    hook_type: HookType,         // TextRender, Dialog, UIElement
    module_name: String,         // "game.exe", "user32.dll"
    retry_count: u32,            // Per gestione fallimenti
}
```

Installazione Hook (semplificato):

```
fn install_hook(address: usize) {
    // 1. Backup bytes originali
    let original_bytes = read_memory(address, 5);

    // 2. Scrivi JMP al nostro handler
    let jmp_instruction = create_jmp_to_handler();
    write_memory(address, jmp_instruction);

    // 3. Salva hook point per cleanup
```

```
    hooks.push(HookPoint {
        address,
        original_bytes,
        // ...
    });
}
```

5. Text Interception

Quando il gioco chiama una funzione hookkata:

```
// Handler chiamato dal nostro hook
fn text_intercept_handler(text: *const u16) {
    // 1. Leggi stringa dalla memoria
    let original_text = read_wide_string(text);

    // 2. Controlla cache
    if let Some(cached) = TRANSLATION_CACHE.get(&original_text) {
        // Cache hit! Inietta immediatamente
        inject_translation(text, &cached.translated);
        return;
    }

    // 3. Cache miss - mostra originale per ora
    // 4. Richiedi traduzione async
    tokio::spawn(async move {
        let translated = translate_text(&original_text).await;
        TRANSLATION_CACHE.insert(original_text, translated);
    });
}
```

6. Translation Flow

Timing & Caching (come hai intuito!):

```
Prima volta:

┌──────────────────────────────────────────────────────────┐
| Gioco chiama DrawTextW("New Game")                       |
└──────────────┬───────────────────────────────────────────┘
```

```
                │
┌───────────────▼──────────────────────────────────────────┐
│ Hook intercetta → Cache miss                             │
└──────────────────────────────────────────────────────────┘
                │
┌───────────────▼──────────────────────────────────────────┐
│ Mostra "New Game" (originale)                            │
└──────────────────────────────────────────────────────────┘
                │
┌───────────────▼──────────────────────────────────────────┐
│ Async: Traduce con LLM → "Nuovo Gioco"                   │
└──────────────────────────────────────────────────────────┘
                │
┌───────────────▼──────────────────────────────────────────┐
│ Salva in cache + Aggiorna display                        │
└──────────────────────────────────────────────────────────┘

Volte successive:
┌──────────────────────────────────────────────────────────┐
│ Gioco chiama DrawTextW("New Game")                       │
└──────────────────────────────────────────────────────────┘
                │
┌───────────────▼──────────────────────────────────────────┐
│ Hook intercetta → Cache HIT (< 1µs)                      │
└──────────────────────────────────────────────────────────┘
                │
┌───────────────▼──────────────────────────────────────────┐
│ Inietta "Nuovo Gioco" ISTANTANEAMENTE                    │
└──────────────────────────────────────────────────────────┘
```

NO freeze del gioco! La traduzione è completamente asincrona.

7. Memory Injection

```rust
fn inject_translation(address: *mut u16, translated: &str) {
    unsafe {
        // Converti stringa in UTF-16 (Windows wide string)
        let wide_string: Vec<u16> = translated.encode_utf16()
            .chain(std::iter::once(0)) // Null terminator
```

```
            .collect();

        // Scrivi nella memoria del processo
        WriteProcessMemory(
            process_handle,
            address as LPVOID,
            wide_string.as_ptr() as *const _,
            wide_string.len() * 2, // bytes
            std::ptr::null_mut()
        );
    }
}
```

8. Translation Bridge

File: `src-tauri/src/translation_bridge/mod.rs`

Server HTTP locale per comunicazione velocissima:

```
// Server in ascolto su 127.0.0.1:random_port
#[tokio::main]
async fn start_bridge_server() {
    let app = Router::new()
        .route("/translate", post(handle_translate))
        .route("/cache", get(handle_cache))
        .route("/stats", get(handle_stats));

    axum::Server::bind(&"127.0.0.1:0".parse()?)
        .serve(app.into_make_service())
        .await?;
}

// Handler traduzione
async fn handle_translate(text: String) ->
Json<TranslationResponse> {
    // Lookup in dictionary (HashMap)
    if let Some(translation) = DICTIONARY.get(&text) {
        return Json(TranslationResponse {
            translated: translation.clone(),
            cached: true,
```

```
            response_time_us: 1, // < 1 microsecondo!
        });
    }

    // Cache miss - richiedi traduzione
    // ...
}
```

Statistiche Real-Time:

```
interface BridgeStats {
  total_requests: number;
  cache_hits: number;
  cache_misses: number;
  errors: number;
  avg_response_time_us: number;  // Microseconds!
  uptime_seconds: number;
}
```

🎮 Multi-Process Support

File: `src-tauri/src/multi_process_injekt.rs` (408 righe)

Alcuni giochi usano architetture multi-processo (es. Chrome-based UI + Game Engine).

```
pub struct MultiProcessConfig {
    pub game_name: String,
    pub primary_process: String,           // "game.exe"
    pub secondary_processes: Vec<String>,  // ["ui.exe",
"launcher.exe"]
    pub injection_strategy: InjectionStrategy,
    pub sync_translations: bool,           // Sincronizza cache tra
processi
    pub max_processes: u32,
}

pub enum InjectionStrategy {
```

```
    PrimaryOnly,        // Inietta solo in game.exe
    AllProcesses,       // Inietta in tutti
    Selective(Vec),     // Inietta in processi specifici
    Cascade,            // Prima primario, poi secondari
}
```

Translation Sync:

- Cache condivisa tra processi
- Quando processo A traduce "Health", processo B lo trova già in cache
- Comunicazione via Translation Bridge

🛡️ Sicurezza & Stabilità

Anti-Cheat Compatibility

Sistemi Rilevati:

- BattlEye (Risk: High)
- Easy Anti-Cheat (Risk: High)
- Vanguard (Risk: Critical)
- VAC (Risk: Medium)
- Denuvo (Risk: Low - solo DRM)
- XIGNCODE3 (Risk: High)
- nProtect GameGuard (Risk: Critical)

Strategie di Bypass:

1. **Delayed Injection**: Aspetta che anti-cheat finisca scan iniziale
2. **Stealth Mode**: Nasconde hooks da scansioni memoria
3. **Safe Window**: Inietta solo durante momenti sicuri (menu, pause)
4. **SetWindowsHook**: Usa API Windows legittime invece di injection diretta

Stabilità

```
// Heartbeat system
const HEARTBEAT_INTERVAL: Duration = Duration::from_secs(5);
const MAX_HOOK_FAILURES: u32 = 10;

// Monitora salute hooks
fn monitor_hooks() {
    loop {
        thread::sleep(HEARTBEAT_INTERVAL);

        for hook in hooks.iter() {
            if !is_hook_valid(hook) {
                hook.retry_count += 1;

                if hook.retry_count > MAX_HOOK_FAILURES {
                    // Hook instabile - rimuovi
                    remove_hook(hook);
                } else {
                    // Tenta reinstallazione
                    reinstall_hook(hook);
                }
            }
        }
    }
}
```

Memory Safety

Rust garantisce memory safety, ma usiamo `unsafe` per WinAPI:

```rust
// Tutti i blocchi unsafe sono documentati e validati
unsafe {
    // SAFETY: process_handle è valido e non null
    // Abbiamo verificato PROCESS_ALL_ACCESS
    WriteProcessMemory(
        process_handle,
        address,
        data,
        size,
        std::ptr::null_mut()
    );
}
```

📊 Performance

Cache Performance

```rust
// HashMap in Rust - O(1) lookup
// Benchmark: ~100ns per lookup (0.0001ms)
let cache: HashMap<String, TranslationCache> = HashMap::new();

// Con 10,000 traduzioni in cache:
// - Lookup: < 1µs
// - Memory: ~2MB
// - Cache hit rate: 95%+ dopo warmup
```

Translation Speed

```
Prima traduzione (cache miss):
- LLM API call: 500-2000ms
- Parsing: < 1ms
- Cache insert: < 1ms
- Total: ~500-2000ms (async, non blocca gioco)

Traduzioni successive (cache hit):
- Cache lookup: < 1µs
- Memory injection: < 10µs
- Total: < 100µs (impercettibile)
```

Memory Footprint

```
GameStringer app:
- Base: ~50MB
- Con cache 10k traduzioni: ~52MB
- Per processo iniettato: +5MB

Confronto:
- Electron app equivalente: ~200MB
- Tauri (Rust): ~50MB
- Risparmio: 75%
```

🎨 UI/UX

Design System

Framework: shadcn/ui (componenti React bellissimi)

Caratteristiche:

- Dark mode nativo
- Animazioni fluide (Framer Motion)
- Responsive design

- Keyboard shortcuts
- Accessibility (ARIA)

Pagine Principali

1. **Library** - Tutti i giochi rilevati
    - Filtri: Piattaforma, Lingua, Backlog
    - Sorting: Nome, Ore giocate, Ultimo avvio
    - Country flags per lingue
2. **Injection Control** - Gestione injection runtime
    - Process selector
    - Hook status real-time
    - Translation stats live
    - Cache management
3. **Translation Memory** - Gestione TM
    - Import/Export
    - Search & filter
    - Batch edit
    - Quality metrics
4. **Settings** - Configurazione
    - API keys (OpenAI, Claude, DeepL)
    - Injection preferences
    - Anti-cheat settings
    - Performance tuning

🔧 Comandi Tauri Principali

Injection Commands

```rust
#[tauri::command]
async fn start_injection(
    process_id: u32,
    process_name: String,
    config: InjectionConfig
) -> Result<SessionInfo, String>

#[tauri::command]
async fn stop_injection(process_id: u32) -> Result<(), String>

#[tauri::command]
async fn get_injection_stats(
    process_id: Option<u32>
) -> Result<InjectionStats, String>
```

Translation Bridge Commands

```rust
#[tauri::command]
async fn translation_bridge_start() -> Result<String, String>

#[tauri::command]
async fn translation_bridge_stop() -> Result<(), String>

#[tauri::command]
async fn translation_bridge_load_translations(
    source_lang: String,
    target_lang: String,
    translations: Vec<TranslationPair>
) -> Result<usize, String>

#[tauri::command]
async fn translation_bridge_get_translation(
    text: String
) -> Result<Option<String>, String>
```

Game Library Commands

```rust
#[tauri::command]
async fn get_steam_games() -> Result<Vec<Game>, String>

#[tauri::command]
async fn get_games_fast() -> Result<Vec<Game>, String>

#[tauri::command]
async fn scan_all_platforms() -> Result<LibraryScanResult, String>
```

🚀 Workflow Completo: Injection Runtime

Esempio Pratico: Tradurre Elden Ring

```javascript
// 1. Utente seleziona gioco dalla library
const game = await invoke('get_game_info', { appId: 1245620 });

// 2. Sistema rileva processo in esecuzione
const processes = await invoke('find_processes');
const eldenRing = processes.find(p => p.name === 'eldenring.exe');

// 3. Anti-cheat detection
const antiCheat = await invoke('detect_anti_cheat', {
    pid: eldenRing.pid
});
// Result: { detected: ['EasyAntiCheat'], risk: 'High' }

// 4. Utente conferma (avviso rischio mostrato)
// 5. Avvia Translation Bridge
await translationBridge.start();

// 6. Carica dizionario pre-tradotto (se disponibile)
await translationBridge.loadFromJson('elden_ring_it.json');
// Loaded: 15,000 translations

// 7. Avvia injection con strategia safe
```

```
const session = await invoke('start_injection', {
    processId: eldenRing.pid,
    processName: 'eldenring.exe',
    config: {
        target_language: 'it',
        provider: 'claude',
        hook_mode: 'safe', // Per anti-cheat
        cache_enabled: true,
    }
});

// 8. Monitor real-time
setInterval(async () => {
    const stats = await invoke('get_injection_stats', {
        processId: eldenRing.pid
    });

    console.log(`
        Hooks attivi: ${stats.active_hooks}
        Traduzioni applicate: ${stats.translations_applied}
        Cache hit rate: ${stats.cache_hit_rate}%
        Uptime: ${stats.uptime_seconds}s
    `);
}, 1000);

// 9. Gioco tradotto in tempo reale!
// - Menu: "New Game" → "Nuova Partita"
// - UI: "Health: 100" → "Salute: 100"
// - Dialog: "You Died" → "Sei Morto"

// 10. Al termine, esporta nuove traduzioni
const newTranslations = await translationBridge.exportToJson(
    'elden_ring_it_updated.json'
);
// Exported: 15,847 translations (+847 nuove)
```

📈 Roadmap & Stato Attuale

✅ Implementato

- ✅ Scan librerie multi-piattaforma
- ✅ Process detection & attachment
- ✅ Anti-cheat detection system
- ✅ Hook infrastructure (WinAPI)
- ✅ Translation Bridge server
- ✅ Cache system
- ✅ Multi-process support
- ✅ UI/UX completa
- ✅ Translation Memory offline

🚧 In Sviluppo

- 🚧 Hook installation completa (alcuni TODO nel codice)
- 🚧 Pattern matching automatico per giochi Unity
- 🚧 Unreal Engine patcher avanzato
- 🚧 Bypass anti-cheat avanzati
- 🚧 Machine learning per rilevamento stringhe UI

🔮 Pianificato

- 🔮 Cloud sync traduzioni
- 🔮 Community translation sharing
- 🔮 OCR per testi renderizzati come texture
- 🔮 Support Linux/macOS (via Wine/Proton)
- 🔮 Plugin system per giochi custom

🎓 Domande Tecniche Risposte

"Come sai a cosa agganciarti per ogni titolo?"

Risposta: Combinazione di approcci:

1. **API Windows Standard**: Tutti i giochi Windows usano DrawTextW, MessageBoxW, ecc.
2. **Pattern Matching**: Cerca pattern comuni in memoria (es. stringhe UI)
3. **Engine-Specific**: Unity usa `UnityEngine.UI.Text`, Unreal usa `FText`
4. **Heuristics**: Analizza chiamate frequenti durante gameplay
5. **Community Profiles**: Database di hook points per giochi popolari

"Il timing - il gioco si blocca?"

NO! Architettura completamente asincrona:

```
Thread 1 (Game):     [Render] → [Hook] → [Cache Hit] → [Inject] → [Continue]
                                              ↓ (miss)
Thread 2 (Async):                  [Translate] → [Cache Update]
```

Il gioco non aspetta mai la traduzione. Prima volta mostra originale, poi aggiorna.

"Injection Unity vs altri engine?"

Unity:

- Injection runtime in `UnityEngine.UI.Text.set_text()`
- Può anche modificare `Assembly-CSharp.dll` offline

Unreal:

- Injection in `FText` rendering
- Patcher per `.pak` files (vedi `unreal_patcher.rs`)

Custom Engine:

- Fallback su API Windows standard
- Pattern matching in memoria

"È legale/sicuro?"

Legale: Sì, per uso personale (single-player)
Sicuro:

- ✅ Single-player: Sicuro
- ⚠️ Multiplayer con anti-cheat: Rischio ban
- ❌ Multiplayer competitivo: NON usare

Il sistema rileva anti-cheat e avvisa l'utente del rischio.

🤝 Beta Testing

Interessato a fare beta testing? Ecco cosa serve:

Requisiti

- Windows 10/11 (64-bit)
- Almeno un gioco installato (Steam/Epic/GOG/ecc.)
- Pazienza per bug e crash 😅

Cosa testare

1. **Library Scanning**: Tutti i giochi vengono rilevati?
2. **Injection Stability**: L'app crasha? Il gioco crasha?
3. **Translation Quality**: Le traduzioni hanno senso nel contesto?
4. **Performance**: FPS drop? Lag?
5. **Anti-Cheat**: Quali giochi funzionano/non funzionano?

Feedback Utile

- Screenshot/video di bug
- Log files (generati automaticamente)
- Specifiche sistema (CPU, RAM, GPU)
- Lista giochi testati con risultati

📚 Risorse

Codice Chiave

- `src-tauri/src/injekt.rs` - Core injection system
- `src-tauri/src/multi_process_injekt.rs` - Multi-process support
- `src-tauri/src/anti_cheat.rs` - Anti-cheat detection
- `lib/injekt-translator.ts` - Frontend injection client
- `lib/translation-bridge.ts` - Bridge client

Documentazione

- Tauri: https://tauri.app/
- WinAPI: https://docs.microsoft.com/en-us/windows/win32/
- Rust: https://www.rust-lang.org/

🎉 Conclusione

GameStringer è un progetto **ambizioso** che combina:

- **Performance** (Rust + WinAPI)
- **Sicurezza** (Memory safety + Anti-cheat detection)
- **UX** (Next.js + shadcn/ui)
- **Innovazione** (Real-time injection + AI translation)

È sia un **tool professionale** per localizzatori che un **traduttore runtime** per giocatori.

Il codice è pulito, ben strutturato, e mostra una profonda comprensione di:

- System programming (WinAPI, memory manipulation)
- Async/concurrency (Tokio, threading)
- Frontend moderno (React, Next.js)
- AI integration (LLM APIs)

Complimenti per il progetto! 🚀

Documento generato il 24 Dicembre 2025
GameStringer - Localizzazione di Videogiochi Potenziata da AI