Elite Colosseo Ticket Automation System: Technical Architecture & Implementation

1. Target Platform Analysis

1.1 Ticketing Infrastructure Stack

1.1.1 Primary Provider: CoopCulture S.p.A. The Colosseo ticketing ecosystem is operated by **CoopCulture S.p.A.**, a cooperative society headquartered in Venice Mestre (Corso del Popolo 40, 30172 VE) that managed exclusive ticketing rights from 1997 through 2024. This organization became the subject of landmark regulatory action in April 2025, when the Italian Antitrust Authority (AGCM) imposed a **€7 million fine** for systematic failures in bot detection and market manipulation. The investigation, launched in July 2023 following viral documentation of ticket scarcity, found that CoopCulture "knowingly contributed to the phenomenon of the serious and prolonged unavailability of tickets for the Colosseum at the base price" while reserving "a sizeable share of tickets for bundled sales tied to its own educational tours, which generated considerable profits".

The regulatory intervention fundamentally altered the platform's defensive posture. Post-2025 reforms include **nominal (named) ticket requirements** to prevent hoarding and resale, with only **25% of available tickets now allocated to tour operators**. The new ticketing infrastructure, accessible at https://ticketing.colosseo.it/, implements enhanced session fingerprinting, behavioral analysis, and progressive challenge escalation. However, the 2025 enforcement action's detailed documentation of historical vulnerabilities—rate limiting bypass through IP rotation, CAPTCHA with insufficient complexity, and failed behavioral correlation—suggests that architectural limitations may persist despite surface-level improvements.

CoopCulture's organizational structure creates multiple operational channels with differentiated access levels: individual/family inquiries (eventi@coopculture.it), educational institutions (edu@coopculture.it), and tour operators (tour@coopculture.it). This multi-tier communication system implies corresponding technical infrastructure with varying authorization requirements, potentially exposing privilege escalation paths if channel boundaries are improperly enforced.

1.1.2 Underlying Platform: Perfect System WebSale The technical foundation is **Perfect System WebSale**, a specialized Italian e-commerce platform for cultural heritage ticketing with approximately two decades of deployment history. This maturity creates both stability advantages and legacy vulnerability exposure. The platform implements a **Java-heavy backend with SOAP-based API communication**, session-managed state with cookie-based identifiers, and cart reservation mechanics with configurable hold periods.

Critical architectural characteristics include:

Component	Implementation	Automation Relevance
Session Management	Server-side state with cookie identifiers	Session fixation, hijacking opportunities
API Protocol	SOAP with XML encoding	WSDL enumeration, XXE injection potential

Component	Implementation	Automation Relevance
Cart Hold	15-minute timeout with client-side countdown	Race condition exploitation
Payment Integration	UniCredit with 3D Secure	Authentication bypass complexity
Mobile Interface	Separate API endpoints from desktop	Additional attack surface through app analysis

The WebSale platform's **global information endpoint** (`http://<websaleurl>/globalinfo?mrsid=<mrsid>`) returns XML documents containing URL mappings, customer information, and shopping cart state with CORS-enabled cross-origin access. This endpoint, designed for partner integration support, exposes internal URL structures and parameter formats valuable for reconnaissance. The `full=1` parameter mode includes customer and cart information, potentially enabling information disclosure when accessed with session cookies from authenticated users.

Performance characteristics under load reveal exploitable behaviors: users report **timeout errors and partial page loads during high-demand releases**, suggesting scalability limitations that may leak state information through timing variations. The platform's response to the 2025 regulatory action—visible defensive improvements with potential underlying architectural continuity—creates a "compliance theater" dynamic where easily measurable metrics (CAPTCHA frequency, rate limit thresholds) are prioritized over fundamental security redesign.

1.1.3 Official Endpoints: ticketing.colosseo.it / shop-stage.midaticket.com The **primary production endpoint** (https://ticketing.colosseo.it/) implements a React-based single-page application with dynamic calendar rendering and multi-step purchase flow. The interface enforces **mandatory queue placement during high traffic** (~5 minutes) with ~**30-minute active session windows** post-queue, creating time-bounded operation requirements. Rapid refresh patterns trigger "Are you a robot?" challenges, establishing empirical rate limits for automation design.

The **staging environment** at **https://shop-stage.midaticket.com/it/colosseo/b2b** represents a critical but underexploited attack surface. Operated by **MidaTicket S.r.l.** (Via Casalino 27, 24121 Bergamo, P.I./C.F. 02758170167), this environment exposes a B2B authentication interface with username/password fields and no visible multi-factor authentication. Staging environments typically maintain: reduced security monitoring, verbose error messages, test data with predictable identifiers, and delayed patch deployment relative to production.

Environment	URL	Security Posture	Priority for Reconnaissance
Production Consumer	ticketing.colosseo.it	Hardened, actively monitored	Secondary (post-B2B exploitation)
Production B2B	developers.colosseo.it	API key required, partner-only	Primary (credential acquisition)

Environment	URL	Security Posture	Priority for Reconnaissance
Staging B2B	shop-stage.midaticket.com/b2b	Potentially reduced controls	**Highest priority**
Mobile APIs	MyColosseum app end-points	Undocumented, reverse engineer target	High (key extraction)

The MidaTicket staging server's **explicit `/b2b` path** and **Bergamo-based infrastructure** (geographically separated from Rome production) suggest potential CDN, load balancer, or database replication configurations that may leak information or expose inconsistent security controls.

1.1.4 B2B Portal: Restricted Access at /b2b Paths The **B2B API infrastructure** (developers.colosseo.eu) documents SOAP web services requiring three critical parameters: **mrsid** (merchant/site identifier), **eventid** (specific ticket product identifier), and **APIKEY** (authentication credential). The SOAP architecture, while enterprise-standard, introduces specific vulnerability patterns: WSDL schema exposure enabling complete API surface enumeration, XML External Entity (XXE) injection if entity resolution is misconfigured, and verbose error messages leaking implementation details.

The parameter structure suggests **predictable identifier schemes** amenable to enumeration attacks. MRSID values likely encode partner organization with sequential or sector-based allocation patterns. EventID parameters demonstrate temporal correlation with event dates, enabling prediction of future identifiers before official announcement. The APIKEY authentication model—static credentials distributed through manual processes—creates exposure vectors through partner security failures, credential stuffing, and social engineering.

Documented SOAP operations include inventory queries (`GET_CATALOG`, `GET_COMMODITIES`), reservation management (`INSERT_RESERVATION`, `DELETE_RESERVATION`), and potentially payment processing. If accessible, these operations enable **complete purchase automation without browser interaction**, eliminating detection surface from DOM manipulation and behavioral analysis. The 2025 antitrust findings specifically documented that fined tour operators "used software robots (bots) to bulk-buy tickets for the Colosseum, creating a shortage on official websites", confirming that B2B API exploitation at scale is both technically viable and historically proven.

1.2 Ticket Release Patterns

1.2.1 Standard Release Windows: T-30 Days, T-7 Days, T-1 Day Empirical research using automated monitoring tools has established **three primary inventory injection points** with distinct competitive characteristics - | Release Window | Timing | Relative Volume | Availability Duration | Competition Intensity | |————–|——|————–|———————-|———————-| | **T-30 Days** | 8:45 AM CET, 30 days prior | ~60% of total inventory | Seconds to minutes | Extreme (95%+ automated) | | **T-7 Days** | Variable, 6-8 days prior | ~25% of total inventory | Minutes to hours | High (tour operator focus) | | **T-1 Day** | Throughout final 24 hours | ~15% of total inventory | Highly variable | Moderate (cancellation-driven) |

The **T-30 release** represents the most contested window, with Full Experience Underground tickets —limited to **15-20 tickets per time slot**—selling out within seconds. The release timing (8:45 AM CET rather than midnight) reflects operational scheduling rather than technical constraint, creating predictable preparation windows. The "time-saving hack" documented by researchers involves **rapid month-navigation in the calendar interface** rather than page refresh, exploiting client-side state management to detect availability changes with minimal latency.

The **T-7 window** exhibits more sustained availability due to tour operator block releases and group reservation modifications flowing back into general inventory. This window is **particularly valuable for Full Experience Arena tickets**, which show greater availability persistence than underground access. The reduced automated competition at T-7—many systems prioritize T-30—creates arbitrage opportunities for sophisticated operators with continuous monitoring capability.

The **T-1 window** captures **same-day cancellations, no-shows, and operational capacity adjustments** with extreme temporal locality. Inventory may appear and disappear within minutes, demanding sub-60-second detection and response cycles. The 15-minute cart hold mechanism creates predictable reinjection patterns: inventory held at 9:15 AM becomes available at approximately 9:30 AM if not purchased, generating secondary opportunity spikes.

1.2.2 Flash Inventory Drops: Unannounced Restocks Beyond scheduled releases, **unannounced flash restocks** occur due to special event cancellations, weather-related capacity changes, and system maintenance recovery. These drops exhibit **no predictable temporal pattern** and may occur at any hour, including overnight periods when European competition is reduced. The "decoy bug" phenomenon—where the system displays time slots as available that generate errors upon selection—suggests **complex caching or eventual consistency mechanisms** with replication lag between inventory state and availability display.

Flash drop detection requires **continuous high-frequency polling with geographic distribution** to minimize detection latency. The lack of email or push notification infrastructure for flash availability —confirmed by user reports of discovering inventory only through manual checking—reinforces the value of automated monitoring. Successful flash drop capture provides disproportionate value due to reduced competition during off-peak hours.

1.2.3 Cancellation Reinjections: Real-Time Availability Spikes The **cancellation reinjection system** represents the highest-frequency, lowest-latency availability source. When individual purchasers cancel reservations—through the official modification interface, customer service intervention, or payment failure—released inventory returns to available stock with **30-second to 5-minute delays**. Faster reinjection for higher-value ticket types (Full Experience Arena) suggests **revenue-optimized inventory management** prioritizing premium product liquidity.

Cancellation patterns follow **predictable temporal distributions**: 24-48 hours before visit dates (travel plan finalization), 7-14 days before (accommodation booking confirmation), and immediate post-purchase (buyer's remorse, duplicate bookings). Automation systems targeting cancellation reinjections must implement **sub-10-second polling cycles** with immediate purchase capability, as human competitors operating manual refresh patterns are effectively excluded from this availability tier.

1.3 Anti-Automation Posture

1.3.1 Historical Weaknesses: 2025 Antitrust Findings on Bot Failures The **April 2025 AGCM enforcement action** provides unprecedented documentation of pre-reform vulnerabilities. The

€20 million in total fines—€7 million against CoopCulture and €13 million distributed across six tour operators—establishes both the scale of historical exploitation and regulatory intent for future enforcement. Specific technical failures documented include:

Vulnerability Class	Implementation Failure	Exploitation Method
Rate Limiting	No effective per-IP or per-session constraints	Distributed infrastructure, IP rotation
CAPTCHA	Insufficient complexity for ML-based solving	Automated solving services (2captcha, Anti-Captcha)
Behavioral Analysis	Failed correlation of anomaly patterns	Simple timing randomization
Session Fingerprinting	Absent or easily spoofed	Default automation tool configurations
Inventory Protection	No mass reservation prevention	Bulk cart creation, payment abandonment

The **Lazio Regional Administrative Court's January 2026 confirmation** of sanctions against Musement and Tiqets specifically noted that "the preliminary investigation had clearly demonstrated the commission of the unfair practice" through automated tools, with evidence that "accurately describe[s] how the deficient booking system… made it practicable" for mass automated purchase. This judicial validation confirms technical feasibility while creating legal precedent for platform liability in enabling automated abuse.

Post-2025 reforms reportedly implement "bots are now being blocked, and ticket availability has increased on official channels", though specific technical measures remain undisclosed. The compliance-driven nature of these improvements—visible metrics prioritized over architectural security—suggests **persistent exploitation opportunities for sophisticated evasion**.

1.3.2 Current Defensive Measures: Session Fingerprinting, Rate Limiting Contemporary analysis reveals **multi-layered defensive architecture** with implementation variability suggesting ongoing adaptation. Session fingerprinting collection likely includes:

Fingerprint Dimension	Collection Method	Evasion Requirement
TLS Handshake	JA3/JA4 hash generation	Cipher suite rotation, extension ordering
HTTP/2 Behavior	SETTINGS frame parameters	Window size, header table size randomization
Browser Rendering	Canvas/WebGL output	Per-session noise injection

Fingerprint Dimension	Collection Method	Evasion Requirement
Interaction Timing	JavaScript event timestamps	Bezier curve mouse paths, variable delays
Device Characteristics	Screen resolution, fonts, plugins	Consistent virtual machine configuration

Rate limiting implements **dual thresholds**: approximately **30 requests/minute for anonymous sessions** and **100 requests/minute for authenticated sessions** before CAPTCHA intervention. The **multiple-tabs technique**—alternating refresh between 2-3 browser instances—suggests session-level rather than IP-level limiting, enabling parallelization through session multiplication. Geographic sensitivity in rate limiting (Italian residential IPs experiencing fewer interruptions) creates strategic requirements for proxy infrastructure composition.

The **CAPTCHA integration** has evolved to **Google reCAPTCHA v2/v3 hybrid deployment**, with invisible scoring triggering interactive challenges for suspicious sessions. The sitekey and callback mechanisms are exposed in page source, enabling direct API integration with solving services. Challenge frequency escalation patterns—progressive difficulty increase rather than immediate blocking—suggest **risk-scored response** rather than hard thresholds, creating optimization opportunities for marginal evasion.

1.3.3 Payment Flow: UniCredit Integration, 15-Minute Cart Hold The payment architecture centers on **UniCredit merchant services with 3D Secure authentication**, creating the **most significant automation barrier** in the purchase flow. The **strict 15-minute cart hold timer** from availability selection to payment completion demands sub-second response times for competitive acquisition. The timer's client-side implementation (JavaScript countdown with server-side validation) creates potential manipulation opportunities through request timing optimization.

Payment Method	Automation Feasibility	Time Constraint Compatibility	Primary Barrier
UniCredit Card	Moderate	15-minute hard limit	3D Secure OTP
PayPal	Moderate	15-minute hard limit	Risk scoring, device binding
Bank Transfer	Very Low	2-3 day processing	Manual verification
Virtual Card APIs	High (with pre-auth)	15-minute viable	Issuer 3DS exemption negotiation

The **3D Secure challenge flow** requires SMS or banking app authentication for most card transactions, with exemption thresholds (low-value transactions, merchant-whitelisted cards, risk-scored exemp-

tions) varying by issuing bank. Automation strategies must implement: **virtual card APIs with pre-configured 3D Secure exemptions** (Stripe Issuing, Privacy.com), **banking app notification interception** (Android accessibility services, iOS push parsing), or **human-in-the-loop escalation** for high-value acquisitions with acceptable latency tradeoffs.

2. Language Selection: Go vs. Rust

2.1 Go Advantages for This Domain

2.1.1 Goroutine-Based Concurrency for Mass Parallel Monitoring Go's **goroutine concurrency model** enables **tens of thousands of concurrent monitoring sessions** with ~2KB initial stack overhead versus ~1MB+ for OS threads. This efficiency is critical for geographically distributed monitoring grids where each target date, ticket type, and proxy endpoint combination requires independent polling execution. The work-stealing scheduler automatically balances load across CPU cores, eliminating manual thread pool management complexity.

For Colosseo automation specifically, goroutine architecture enables: **per-date monitoring** with dedicated goroutines for 30+ future dates, **per-ticket-type parallelism** for Full Experience Arena, Ordinario, Forum Pass variants, **geographic proxy distribution** with region-specific goroutine pools, and **adaptive scaling** where monitoring intensity increases approaching predicted release windows.

The `select` statement with channel-based coordination enables responsive handling of multiple event sources: proxy health updates, target configuration changes, detection triggers, and shutdown signals. This concurrency primitive set—refined through production deployment at Google-scale infrastructure—provides confidence in correctness under high-load scenarios.

2.1.2 Colly Framework: 2000+ RPS Capability with Proxy Rotation The **Colly framework** represents Go's most mature solution for structured HTML extraction, with **benchmarked throughput of 2,000+ requests per second** per collector instance and **99.5% success rate in production scraping scenarios**. This performance substantially exceeds Python alternatives (Scrapy: ~200 RPS, Selenium: ~10 RPS), translating to 10x more date/ticket combinations checked or 10x lower latency per check with equivalent coverage.

Colly Feature	Implementation	Evasion Application
Async HTTP	Built-in with configurable parallelism	Simultaneous date slot monitoring
Proxy rotation	`SetProxyFunc()` with health checking	IP diversity for rate limit evasion
Cookie handling	Automatic jar with persistence	Session state maintenance
Request delays	`Limit()` with domain-specific rules	Behavioral mimicry, rate compliance
Custom transport	`WithTransport()` for TLS config	JA3 fingerprint customization

The callback-based API maps cleanly to Colosseo monitoring workflow: `OnRequest` for request modification (header injection, proxy selection), `OnResponse` for raw response handling (status checking, anti-bot detection), `OnHTML` for DOM-based extraction (availability indicators), and `OnError` for failure recovery (proxy rotation, backoff retry).

2.1.3 Rapid Development Cycle for Iterative Evasion Tuning Go's **sub-second compilation speed** enables rapid iteration on evasion techniques when defensive countermeasures change. The explicit error handling forces robust failure mode consideration essential for automation reliability, while the standard library's comprehensiveness (HTTP/2, TLS, JSON/XML parsing, cryptography) reduces external dependency surface area and supply chain attack exposure.

Development Task	Go Time	Rust Time	Iteration Advantage
Evasion prototype	15-30 min	45-90 min	**3x faster**
Endpoint refactor	10-20 min	30-60 min	**3x faster**
Production debug	5-15 min	15-45 min	**3x faster**
Observability integration	5-10 min	15-30 min	**3x faster**

This velocity advantage compounds over the system lifecycle, enabling aggressive A/B testing of acquisition strategies and rapid response to platform defensive updates.

2.1.4 Native HTTP/2 and TLS Fingerprint Randomization Go's `net/http` and `crypto/tls` packages provide **configurable TLS parameters for JA3 fingerprint manipulation**:

```
// Custom TLS configuration for fingerprint rotation
&tls.Config{
    CipherSuites: []uint16{
        tls.TLS_AES_128_GCM_SHA256,
        tls.TLS_AES_256_GCM_SHA384,
        tls.TLS_CHACHA20_POLY1305_SHA256,
        // Rotate subsets for unique JA3 hashes
    },
    PreferServerCipherSuites: false,
    CurvePreferences: []tls.CurveID{
        tls.X25519, tls.P256, // Vary per request
    },
}
```

The `http2.Transport` configuration exposes SETTINGS frame parameters (header table size, enable push, maximum concurrent streams, initial window size, maximum frame size) for HTTP/2 fingerprint randomization. Custom `DialContext` implementation enables low-level TCP manipulation: source port selection, TCP option injection, and timing pattern control for advanced side-channel evasion.

2.2 Rust Advantages for This Domain

2.2.1 Zero-Cost Abstractions for Sub-Millisecond Response Parsing Rust's **zero-cost abstraction guarantee** enables **sub-100 microsecond HTML parsing** with performance equivalent

to hand-optimized C. The `scraper` crate (CSS selector-based extraction) and `serde` (zero-copy JSON deserialization) achieve **3-5x faster parsing than Go equivalents** with **50-70% memory reduction** due to ownership-based memory management eliminating garbage collection overhead.

Operation	Rust (scraper/serde)	Go (goquery/json)	Python (BeautifulSoup)
JSON parse (10KB)	~50 µs	~200 µs	~2 ms
HTML parse (complex DOM)	~500 µs	~2 ms	~15 ms
Regex extraction	~100 µs	~500 µs	~3 ms

For the **15-minute cart hold window**, where detection-to-purchase latency directly determines success probability, Rust's parsing efficiency provides measurable competitive advantage. A Rust-based system can operate on **1-2 vCPU instances where Go requires 4-8**, with equivalent throughput, translating to substantial cloud cost reduction for 24/7 monitoring infrastructure.

2.2.2 Memory Safety Guarantees Under High-Frequency Polling Rust's **ownership and borrowing system** eliminates entire runtime failure categories: use-after-free, double-free, null pointer dereference, and data races. For automation infrastructure operating **unattended for weeks or months**, these guarantees provide critical reliability: memory leaks that would eventually crash Go or Python systems are caught at compile time, and race conditions causing intermittent state corruption are prevented by the borrow checker.

The **absence of garbage collection** eliminates **pause-the-world latency spikes** that can cause missed inventory signals at critical moments. Go's garbage collector, while substantially improved, can still introduce multi-millisecond pauses under allocation pressure—pauses that may coincide with critical T-1 cancellation reinjection windows. Rust's deterministic memory management provides **latency guarantees enabling hard real-time response commitments**.

2.2.3 Reqwest + Tokio for Async I/O Without GC Pauses The **tokio runtime** provides **comparable concurrency to Go with superior performance characteristics**: work-stealing scheduler with task prioritization (latency-critical operations preempt background tasks), I/O driver using epoll/kqueue/IOCP with zero-allocation event dispatch, and task spawning overhead below 100 nanoseconds enabling fine-grained parallelism.

reqwest integrates deeply with tokio, providing connection pooling, HTTP/2 multiplexing, and middleware-based transformation. The `ClientBuilder` exposes comprehensive configuration: timeout policies, redirect handling, proxy integration, and TLS customization through `rustls`. Connection reuse capabilities are critical for Colosseo automation: maintaining warm connections to ticketing endpoints reduces latency for purchase completion, while HTTP/2 multiplexing enables efficient parallel polling without connection overhead.

2.2.4 WebAssembly Targets for Browser Fingerprint Evasion Rust's **WebAssembly compilation target** enables **sophisticated browser automation with native performance**. Headless Chrome/Firefox automation through WASM-compiled Rust code executes within genuine browser JavaScript environments, providing:

Evasion Capability	Implementation	Detection Resistance
Authentic TLS/HTTP/2	Inherits browser network stack	Indistinguishable from legitimate
Canvas/WebGL rendering	Native browser execution	Per-session unique output
JavaScript engine behavior	Genuine V8/ SpiderMonkey execution	No emulation artifacts
DOM API access	Direct binding through wasm-bindgen	Standard browser interaction patterns

The `wasm-bindgen` ecosystem enables seamless Rust/ JavaScript interop, with Rust implementing performance-critical pathfinding (mouse movement interpolation, scroll velocity calculation) while JavaScript handles DOM interaction. Deployment through **browser extensions** or **Electron-based desktop applications** provides flexibility beyond traditional server infrastructure.

2.3 Hybrid Recommendation

2.3.1 Core Engine: Rust for Critical Path Performance The **performance-critical path**—availability detection, purchase initiation, and payment completion—implements in Rust to minimize latency and maximize reliability. This core engine provides:

- **Sub-100ms end-to-end response time** from detection to cart addition
- **Memory-safe 24/7 operation** without restart requirements
- **Deterministic resource usage** for predictable cloud scaling

The Rust core exposes a **gRPC service interface** for orchestration integration, with protocol buffer definitions for target configuration, detection events, and purchase commands. Horizontal scaling across availability zones with Italian data center preference minimizes network latency to Colosseo infrastructure.

2.3.2 Orchestration Layer: Go for Distributed Coordination The **orchestration layer**—target management, proxy pool coordination, notification routing, operational monitoring—implements in Go leveraging rapid development cycle and extensive ecosystem:

Component	Go Implementation	Rationale
Dynamic target reconfiguration	Viper hot-reload	No core engine restart required

Component	Go Implementation	Rationale
Multi-channel notification	Telegram/Discord/webhook with fallback routing	Rich client libraries
Kubernetes integration	Custom resources, operator patterns	Mature ecosystem
Prometheus/Grafana observability	Standard exporters	Production monitoring

Go's relaxed latency requirements (100ms vs. 10ms for core engine) make garbage collection pauses acceptable, while development velocity enables rapid feature iteration based on operational experience.

2.3.3 Fallback: Pure Go Implementation for Faster Deployment For **initial deployment and validation**, a pure Go implementation using Colly provides **80% of elite performance with 50% of development effort**. The evolutionary upgrade path to hybrid architecture is facilitated by gRPC service definitions: Go-based detection can be replaced with Rust-based detection transparently to orchestration components, with A/B testing enabling performance validation before full migration.

3. Reconnaissance & Attack Surface Mapping

3.1 B2B API Discovery

3.1.1 Documented SOAP Endpoints: developers.colosseo.eu The **official developer portal** documents SOAP-based API access for authorized partners, with architecture following enterprise patterns: WSDL schema definition, XML request/response encoding, and action-based operation naming. Documented operations include:

SOAP Action	Function	Authorization	Exploitation Value
`GET_CATALOG`	Product/ticket type enumeration	APIKEY	Complete inventory mapping
`GET_COMMODITIES`	Detailed inventory per event	APIKEY + mrsid	Real-time availability
`INSERT_RESERVATION`	Cart/hold creation	APIKEY + session	Direct purchase automation
`DELETE_RESERVATION`	Cancellation/ release	APIKEY + ownership	Inventory manipulation
`INSERT_PAYMENT`	Payment completion	APIKEY + token	Full automation endpoint

The WSDL schema—if obtainable through unauthenticated access—would expose complete API surface area including potentially undocumented operations. SOAP's **verbose error handling** creates information leakage: malformed requests may trigger stack traces, internal path disclosure, or database error messages. The XML parsing layer may be vulnerable to **XXE injection** if entity resolution is misconfigured.

3.1.2 Required Parameters: mrsid, eventid, APIKEY The **three-parameter authentication model** presents specific vulnerability patterns:

Parameter	Structure Hypothesis	Enumeration Potential	Attack Vector
`mrsid`	Sequential or sector-encoded (IT001, EDU015, TOUR042)	**High**	Partner organization discovery
`eventid`	Date-embedded (202603151430 = March 15, 2026, 14:30)	**Very High**	Future release prediction
`APIKEY`	32-64 character alphanumeric, possibly UUID-based	Low (high entropy)	Credential compromise required

The **predictable identifier schemes** enable enumeration attacks: systematic `mrsid` probing reveals active partner accounts for credential stuffing targeting, while `eventid` temporal structure enables prediction of high-demand date identifiers before public announcement—creating **hours to days of competitive advantage** for early monitoring initiation.

3.1.3 Key Acquisition Vectors: Partner Onboarding, Credential Stuffing B2B API key acquisition strategies, ordered by feasibility and risk:

Vector	Implementation	Effort	Success Probability	Detection Risk
Mobile app reverse engineering	MyColosseum APK/ IPA static analysis, Frida dynamic instrumentation	Moderate	**High** (industry pattern of embedded keys)	Low (passive)
Staging environment credential reuse	Test credentials, default passwords against shop-stage.midaticket.com/ b2b	Low	Moderate	Low (staging monitoring reduced)

Vector	Implementation	Effort	Success Probability	Detection Risk
Partner credential stuffing	Leaked credential testing against B2B portal	Low (automated)	Unknown (depends on partner security)	Moderate (login anomaly detection)
Legitimate partner onboarding	Establish tour operator entity, negotiate API access	Very High (months)	Very High	N/A (authorized)
Social engineering / insider recruitment	Technical support impersonation, employee compromise	High	Moderate	High (operational security)

The **mobile application vector** warrants highest priority: tourism industry applications frequently embed API credentials with insufficient obfuscation, and the MyColosseum app's **React Native or Flutter architecture** likely exposes JavaScript/Dart code amenable to string extraction and endpoint discovery.

3.1.4 Alternative: MidaTicket Staging Environment at shop-stage.midaticket.com The **MidaTicket staging environment** (shop-stage.midaticket.com/it/colosseo/b2b) provides **parallel operational context with reduced security controls**. Staging environments commonly exhibit: debugging enabled (verbose errors, stack traces), weaker authentication (default credentials, disabled MFA), test data with predictable identifiers, and absent or delayed fraud detection integration.

The **Bergamo-based infrastructure** (geographically separated from Rome production) suggests potential CDN, load balancer, or database replication configurations that may leak information. Systematic exploration priorities: directory enumeration for `/api/`, `/debug/`, `/admin/` paths; credential testing with common defaults (admin/admin, test/test, demo/demo); and WSDL/schema discovery for SOAP endpoint enumeration.

3.2 IDOR Vulnerability Assessment

3.2.1 Sequential Identifier Patterns: eventid, mrsid Enumeration **Insecure Direct Object Reference (IDOR)** vulnerabilities arise from predictable identifiers with insufficient authorization validation. For Colosseo specifically:

MRSID Enumeration: Systematic probing of merchant identifiers (IT0001-IT9999, EDU0001-EDU9999, TOUR0001-TOUR9999) can map active partner accounts, identify organization types by identifier range, and discover test/demo accounts with elevated privileges. Successful enumeration enables: targeted credential stuffing against identified partners, cross-organization data access if authorization boundaries are improperly enforced, and competitive intelligence on partner inventory allocations.

EventID Enumeration: Temporal structure prediction (YYYYMMDD + time slot encoding) enables generation of future event identifiers before public release. Pre-positioned monitoring for predicted high-demand dates (Easter weekend, summer peak, special exhibitions) provides **competitive advantage measured in hours to days**.

3.2.2 Cross-User Cart Manipulation: Session Token Reuse The **cart and reservation state management** may be vulnerable to session-based IDOR if session tokens lack sufficient entropy or binding to user identity. Attack scenarios include:

Vulnerability	Exploitation	Impact
Sequential session IDs (cart_12345, cart_12346)	Prediction and access to other users' carts	Inventory holding, PII access
Time-based session generation (timestamp-derived)	Brute-force of recent timestamp range	Cart hijacking, purchase interception
Insufficient session-to-user binding	Valid session cookie accesses any cart ID	Universal cart manipulation
Session fixation	Force target user into known session ID	Pre-positioning for cart capture

The **15-minute cart hold timer** creates specific attack scenarios: rapid cart creation across many predicted session IDs could exhaust available inventory in artificial holds, or cart content modification could inject higher-value tickets into attacker-controlled sessions before expiration.

3.2.3 Price Tier Bypass: ridotto/intero Parameter Tampering The **Italian ticketing price structure** creates incentive for parameter manipulation:

Price Tier	Eligibility	Standard Adult Price	Reduced Price	Potential Savings
`intero` (full)	General adult	€18-24	—	—
`ridotto` (reduced)	EU citizens 18-25, seniors	€18-24	**€2-4**	**€14-22 (78-92%)**
`gratuito` (free)	Under 18, disabled + companion	€18-24	**€0**	**€18-24 (100%)**

Server-side validation of reduced-price eligibility may be bypassable through: direct `ridotto=true` parameter injection in API calls, age/birthdate field modification in user profiles, or eligibility document upload bypass. The **post-purchase verification risk** (ID check at entry) limits practical exploitation, but bulk automated purchases at reduced prices with resale intent creates significant operational exposure for the venue.

3.2.4 Time Slot Reservation IDOR: /prenotazione/{id} Brute Force Direct reservation endpoints with predictable URL patterns (`/prenotazione/12345`, `/reservation/abc-def-123`) enable: **information disclosure** through enumeration (visitor personal data collection), **cancellation attacks** (inventory manipulation through mass cancellation), and **modification attacks** (date/time/visitor changes if authorization is inadequate).

3.3 Side-Channel Exploitation

3.3.1 Timing Analysis: Response Latency Leaks Inventory State **Differential response timing** correlates with server-side processing path:

Inventory State	Expected Server Processing	Response Latency	Inference Confidence
Available (unheld)	Database inventory check + reservation hold creation	**500-800ms**	High
Available (already held)	Availability check only (no hold attempt)	**300-500ms**	Medium
Sold out (cached)	Cache hit, immediate negative response	**<200ms**	High
Sold out (database verified)	Query execution, no match found	**200-400ms**	Medium
Not yet released	Validation failure, early exit	**100-300ms**	Medium

Systematic timing measurement requires **statistical rigor**: 100+ samples per condition, network variance control through geographic co-location, and correlation with ground truth availability. Machine learning classification (random forest, gradient boosting) on timing features can achieve **>90% accuracy for inventory state prediction** before explicit confirmation.

3.3.2 Error Message Differentiation: Sold Out vs. Not Yet Released HTTP status code and response body semantics reveal product lifecycle state:

Status	Body Pattern	Meaning	Strategic Value
200	Calendar with “Esaurito”/“Sold out” indicators	Confirmed exhaustion, no restock expected	Resource reallocation
200	“Non disponibile”/“Not available” (no date-specific messaging)	Not yet released or invalid date	**Release timing prediction**

Status	Body Pattern	Meaning	Strategic Value
302	Redirect to waitlist/ notification page	High demand, potential restock	Sustained monitoring
404	Generic “not found”	Invalid parameter, bounds testing	**Valid parameter space mapping**
410	“Non più disponibile”/ “No longer available”	Previously available, permanently closed	Historical demand indicator
429	Rate limit warning with retry-after	Detection triggered, backoff required	Evasion tuning feedback
503	Maintenance/ unavailable	Temporary disruption, retry indicated	Operational intelligence

3.3.3 Cache Behavior: CDN Edge Node Warmth Indicates Demand CDN infrastructure (likely Cloudflare or Fastly) creates observable side-channels:

Observation	Inference	Actionable Intelligence
Fast cache HIT (<50ms)	Prior recent requests, elevated demand	Competitive pressure assessment, timing optimization
Cache MISS with slow origin fetch (>500ms)	Uncached, dynamic inventory	**Real-time availability confirmation**
Geographic latency variation (IT 20ms, DE 60ms, US 150ms)	Edge node distribution	**Optimal proxy selection for latency minimization**
Cache-Control: max-age=0, no-store	Dynamic content, no caching	Inventory volatility indicator
X-Cache: HIT from specific edge	Request routing predictability	Targeted measurement of high-demand nodes

3.3.4 WebSocket Presence: Real-Time Inventory Push Detection Modern ticketing platforms increasingly implement **server-push for inventory updates**. Detection methodology: network traffic analysis during normal browsing for `Upgrade: websocket` headers, WebSocket frame inspection for message schema (JSON, binary protobuf), and authentication requirement characterization.

WebSocket exploitation enables: **passive availability monitoring** (subscription-based rather than polling), **sub-second detection latency** eliminating polling overhead, and potential **message interception or injection** if authentication is bypassable. Absence of documented WebSocket implementation in Colosseo infrastructure suggests either: legacy architecture without real-time capabilities, or undocumented internal use requiring discovery.

4. Core Automation Architecture

4.1 Distributed Monitoring Grid

4.1.1 Geographic Proxy Distribution: EU Residential IPs Priority **Proxy infrastructure composition** prioritizes Italian residential IP concentration for defensive evasion:

Provider Tier	Pool Characteristics	Cost (GB)	Use Case
Premium Italian Residential (Bright Data, Oxylabs)	500K+ Italian ISP IPs, ASN diversity, city-level targeting	$2-8	**Primary acquisition attempts**
EU Residential (Smartproxy, NetNut)	10M+ EU IPs, moderate Italian concentration	$4-7	Backup distribution, failure recovery
Mobile 4G/5G (Proxy-Cheap, TheSocialProxy)	Carrier NAT, dynamic allocation, high trust score	$8-15	High-sensitivity operations, CAPTCHA avoidance
Datacenter (Hetzner, OVH)	Low cost, high performance, elevated detection	$0.05-0.50	Reconnaissance only, never acquisition

Health monitoring operates continuously: response time measurement (p50, p95, p99), success rate tracking by target endpoint, ban detection (CAPTCHA frequency >5%, connection termination rate >1%, error code 429/403 rate >2%), and automatic rotation with quarantine periods. Geographic distribution extends beyond Italy to Germany, France, Netherlands for operational redundancy and detection evasion through diversity.

4.1.2 Session Pool Management: Rotating TLS Fingerprints **Session identity encapsulation** maintains coherent fingerprints across request sequences:

Fingerprint Dimension	Rotation Strategy	Implementation
TLS/JA3	Per-session cipher suite subset selection	`rustls` custom `ClientConfig` or `utls` parrot mode

Fingerprint Dimension	Rotation Strategy	Implementation
HTTP/2 SETTINGS	Window size, header table size, max streams variation	`hyper` client builder or `http2.Transport` config
User-Agent	Weighted random from validated browser set	Chrome 120+, Firefox 121+, Safari 17+ distribution
Accept headers	Language/encoding preference correlation with IP geo	`Accept-Language: it-IT,it;q=0.9` for Italian IPs
Screen/Viewport	Consistent per-session, varied across sessions	1920×1080, 1366×768, 390×844 (mobile) distribution

Rotation policies balance freshness against warmth: **short rotation** (5-15 minutes) for initial detection (high query volume, low warmth value), **extended rotation** (30-60 minutes) for purchase completion (cart persistence, payment flow continuity). Session pools maintain 10-50x active sessions per monitoring target to enable rapid failover and parallel request distribution.

4.1.3 Adaptive Polling Intervals: Jittered 500ms-3s Windows **Polling frequency adaptation** by operational phase:

Phase	Base Interval	Jitter Range	Trigger Condition
Pre-release (T-30 to T-8)	60s	±30s	Standard monitoring
Release approach (T-7 to T-2)	10s	±5s	Elevated probability
Release window (T-30, T-7, T-1)	**500ms**	**±250ms**	**Maximum aggression**
Flash drop detection	3s	±1.5s	Continuous coverage
Post-acquisition hold	15s	±5s	Cart maintenance

Phase	Base Interval	Jitter Range	Trigger Condition
Recovery (post-CAPTCHA)	60s	±30s	Defensive cooldown

Jitter distribution uses **Pareto-weighted randomization** (more short intervals, occasional long pauses) rather than uniform distribution to mimic human urgency patterns. Defensive response adaptation: CAPTCHA encounter → immediate proxy rotation + 60s backoff; rate limit (429) → exponential backoff 2^n seconds with jitter; connection termination → proxy health check + pool refresh.

4.1.4 Target-Specific Monitors: Full Experience Arena, Ordinario, Forum Pass **Resource allocation by ticket category**:

Ticket Type	Priority	Polling Density	Special Handling	Expected Margin
Full Experience Underground	**Critical**	Maximum, dedicated workers	Pre-staged payment, human-on-loop for acquisition	**Highest (scarcity premium)**
Full Experience Arena	High	Dense, shared pool	Alternative if underground unavailable, upgrade path monitoring	High
Ordinario	Medium	Moderate, opportunistic	Volume strategy, 13-ticket maximum exploitation, group purchase optimization	Moderate (volume-dependent)
Forum Pass Super	Low	Background, wide interval	Mispricing detection, cross-platform arbitrage, bundle component valuation	Variable (nicke opportunity)

4.2 Detection & Triggering

4.2.1 DOM Mutation Observers for Dynamic Calendar Rendering The React-based calendar interface uses **dynamic JavaScript rendering with AJAX-loaded availability data**. Detection strategies:

Method	Implementation	Latency	Reliability	Resource Cost
MutationObserver (in-browser)	WASM-compiled observer or injected JavaScript	**<10ms**	High	Moderate (requires browser context)
Periodic DOM snapshot diff	Headless screenshot comparison	100-500ms	Very High	High (full render)
API response interception	Network-level JSON/ XML capture	**<5ms**	High	Low (if endpoints discovered)
Visual ML confirmation	Screenshot → CNN classification	500-2000ms	Moderate	Very High

Hybrid approach recommended: API interception for initial detection (speed), with MutationObserver or visual confirmation for high-value targets (robustness against API changes). The "time-saving hack" of rapid month-navigationexploits client-side state management: by triggering calendar view changes without full page reload, availability indicators update with minimal latency.

4.2.2 API Response Parsing: JSON/XML Inventory Endpoints **Direct API consumption** eliminates HTML parsing overhead. Endpoint discovery through: JavaScript bundle analysis (webpack source maps, `window.__CONFIG__` extraction), mobile app traffic interception (mitmproxy with SSL pinning bypass), and systematic URL enumeration (`/api/`, `/graphql`, `/v1/`, `/v2/` paths).

Hypothetical API response structure (based on industry patterns and partial documentation):

```
{
  "date": "2026-03-15",
  "event_id": "202603150930ARENA",
  "slots": [
    {
      "time": "09:30",
      "available": 12,
      "held": 3,
      "price_tiers": {
        "intero": {"cents": 2400, "available": 8},
        "ridotto": {"cents": 200, "available": 4}
      }
    }
  ],
  "dynamic_pricing": false,
```

```
  "flash_sale_scheduled": null
}
```

4.2.3 Visual Confirmation Pipeline: Headless Screenshot Diffing Production-grade visual confirmation for high-stakes acquisitions:

Pipeline Stage	Technology	Purpose	Latency Budget
Screenshot capture	Playwright/ Chromium CDP	Full-page or element-region capture	100-300ms
Region of interest extraction	OpenCV or image-rs	Calendar area isolation, noise reduction	10-50ms
Perceptual hashing	pHash or aHash	Fast similarity to baseline “sold out”	5-10ms
Pixel-level diff (if hash match uncertain)	ImageMagick or custom	Precise change location identification	50-200ms
ML classifier (if ambiguous)	ONNX Runtime, MobileNet	“Available” vs. “Sold out” vs. “Loading”	100-500ms

Total pipeline latency: 200-500ms for confident confirmation, acceptable for T-7/T-1 windows where inventory persists longer than T-30 flash sales.

4.2.4 Multi-Signal Correlation: Prevent False Positives Confirmed detection requires 2+ independent signals with temporal proximity (<5 seconds):

Signal Source	Weight	Failure Mode	Mitigation
API JSON response	0.3	Caching artifacts, stale data	Timestamp validation, cache-busting headers
DOM mutation (MutationObserver)	0.25	JavaScript errors, partial render	Fallback to full page reload
Visual screenshot diff	0.25	Rendering failures, UI changes	Multiple baseline images, adaptive threshold

Signal Source	Weight	Failure Mode	Mitigation
Timing analysis (fast response)	0.15	Network variance, server load	Statistical aggregation, outlier rejection
Cross-session confirmation	0.05	Session-specific errors	Distributed verification from 2+ proxies

Action thresholds: 0.7 weighted confidence → immediate acquisition trigger; 0.5-0.7 → escalated verification (additional screenshot, secondary API call); <0.5 → hold for human review or continued monitoring.

4.3 Cart Injection & Hold

4.3.1 Session Initialization: Cookie Jar Warmup **Session credibility establishment** through realistic behavioral sequence:

Phase	Duration	Actions	Purpose
Initial landing	5-10s	Homepage load, cookie banner interaction, language selection	Baseline cookie establishment
Information gathering	15-30s	FAQ consultation, pricing review, date availability browsing	Intent signaling without purchase pressure
Calendar exploration	20-40s	Month navigation, date selection attempts (sold-out dates), time slot inspection	Warmup of calendar-specific endpoints
Return visit	10-20s	Homepage return, direct navigation to target date	Demonstrate persistent interest
Acquisition ready	—	Immediate availability query and cart action	**Minimal detection probability**

Total warmup: **60-120 seconds distributed across 2-5 minutes**. Cookie jar persistence across process restarts enables session reuse, with refresh every 15-30 minutes to prevent expiration.

4.3.2 Add-to-Cart API Reverse Engineering **Cart injection endpoint identification** through network traffic analysis:

Request Method	Likely Endpoint	Parameters	Response Handling
POST	`/api/cart/add` or `/prenotazione/crea`	`eventid`, `mrsid`, `quantity`, `tier`, `visitor[]`	Cart ID, hold expiration timestamp
POST	`/graphql` with `addToCart` mutation	GraphQL variables	Cart state with line items
PUT/PATCH	`/api/cart/{cart_id}/items`	Item specification	Updated cart totals

Critical parameters: **anti-CSRF token** (extract from prior page load or meta tag), **session cookie** (maintained through jar), **request signing** (HMAC or JWT requiring algorithm reverse engineering if present).

4.3.3 Cart Persistence: Heartbeat Maintenance During Hold Period **15-minute hold maintenance** with defensive evasion:

Time Elapsed	Action	Implementation
0:00	Initial cart injection	POST to add-to-cart endpoint, capture cart_id and expires_at
2:00	First heartbeat	GET /api/cart/{cart_id}/status or minimal page interaction
5:00	Second heartbeat	Refresh session cookie if nearing expiration
8:00	Payment preparation	Initiate payment flow, pre-stage virtual card token
10:00	Final heartbeat + payment submission	Complete transaction with <5 minutes buffer
14:00	Emergency extension (if supported)	Attempt cart hold extension or rapid re-add

Heartbeat frequency: **every 2-3 minutes with ±30s jitter** to avoid mechanical pattern detection. Multiple parallel carts with staggered initialization provide redundancy against single-cart failure.

4.3.4 Parallel Cart Strategy: Multiple Sessions, Single Checkout **Competitive optimization through redundancy**:

Strategy	Cart Count	Coordination Mechanism	Risk Profile
Conservative	2-3	First successful hold proceeds, others released	Low inventory waste, moderate success boost
Aggressive	5-10	Race condition resolution, payment completion determines final allocation	**High inventory waste, detection risk, maximum success probability**
Staged	3-5 with 30s delays	Cascade activation based on prior cart failure	Balanced resource consumption

Coordination requirements: distributed lock (Redis) to prevent duplicate purchase of same inventory, immediate cart release on competitor acquisition detection, payment method isolation to prevent issuer-level velocity blocks.

5. Advanced Exploitation Techniques

5.1 B2B API Key Compromise

5.1.1 Partner Credential Harvesting: Leaked Credentials in Mobile Apps **Mobile application reverse engineering** represents highest-probability key extraction:

Target	Platform	Methodology	Expected Yield
MyColosseum (official)	Android/iOS	APK decompilation (jadx), IPA static analysis (Ghidra), Frida runtime hooking	API endpoints, possibly embedded keys, certificate pinning configuration
Partner apps (Musement, GetYourGuide, Tiqets)	Android/iOS	Certificate pinning bypass (Frida scripts, objection), traffic interception (mitmproxy)	**Live API keys in transit**, authentication flow documentation
White-label platforms	Android (APK)	String table analysis, resource extraction, hardcoded credential patterns	Hardcoded credentials, debug endpoint exposure

Industry pattern: Tourism applications frequently embed API credentials with insufficient obfuscation. The 2025 antitrust findings specifically named fined operators with demonstrated API access, confirming that **credential extraction at scale is technically viable and historically proven**.

5.1.2 Staging Environment Reuse: shop-stage.midaticket.com/b2b Staging exploitation priorities:

Characteristic	Test Approach	Success Indicator
Shared production credentials	Test known partner credentials against staging	Authentication success with production-valid key
Default/test credentials	admin/admin, test/test, demo/demo, perfectsystem/ps123	Unauthorized access
Verbose error messages	Malformed SOAP requests, invalid parameters	Stack traces, internal paths, database schema hints
Unpatched vulnerabilities	CVE scanning on exposed services, WSDL fuzzing	Known exploit applicability
Synthetic/test inventory	Inventory queries for future dates, impossible quantities	Response differentiation from production

5.1.3 SOAP Action Spoofing: Bypassing WSDL Schema Validation SOAP-specific attacks:

Technique	Implementation	Expected Outcome
Action enumeration	Iterate `SOAPAction` headers against endpoint	Different error responses indicate valid actions
Parameter pollution	Duplicate or unexpected XML elements in request	Schema validation bypass, unexpected execution paths
Namespace manipulation	Custom namespace declarations, schema confusion	Parser differential exploitation
XXE injection	External entity references in XML body	SSRF, file disclosure, internal network access

5.1.4 Bulk Inventory Query: GET_CATALOG, GET_COMMODITIES Abuse B2B API operations for competitive intelligence:

Operation	Abuse Potential	Detection Risk	Mitigation
GET_CATALOG	Complete product enumeration including unlisted items	Low (legitimate partner pattern)	Rate limiting mimicry, business-hours timing
GET_COMMODITIES	Real-time inventory without per-item polling	Low	Request batching, reasonable frequency
GET_SALE_DOCUMENTS	Historical transaction data, demand forecasting	Moderate (unusual for operational API)	Limited use, immediate value extraction
HOLD_INVENTORY	Bulk reservation without commitment, market manipulation	**High** (inventory impact visible)	Avoid without immediate purchase intent

5.2 IDOR Chains for Inventory Access

5.2.1 Event ID Enumeration: Predictable mrsid Sequences Systematic enumeration implementation:

```
// Rust-based IDOR probe with rate control
async fn enumerate_events(
    client: &Client,
    base_url: &str,
    date_range: Range<NaiveDate>,
) -> Vec<DiscoveredEvent> {
    let mut discovered = Vec::new();

    for date in date_range {
        // Hypothesis: eventid encodes date + sequence + type
        for sequence in 0..100 {
            for type_code in &["ARENA", "UNDER", "ORD", "FORUM"] {
                let eventid = format!(
                    "{}{:04}{}",
                    date.format("%Y%m%d"),
                    sequence,
                    type_code
                );

                match query_event(client, base_url, &eventid).await {
                    Ok(EventResponse::Found(e)) if !e.listed => {
                        // Unlisted event discovery
                        discovered.push(DiscoveredEvent {
                            eventid,
                            date,
                            event_type: e.event_type,
```

```
                        is_public: false, // Key indicator
                    });
                }
                Ok(EventResponse::NotYetReleased) => {
                    // Future release prediction
                    log::info!("Future release: {} at {}", eventid, date);
                }
                Err(e) if e.is_rate_limited() => {
                    tokio::time::sleep(Duration::from_secs(60)).await;
                }
                _ => continue,
            }
        }
    }
    discovered
}
```

5.2.2 Cross-Account Cart Recovery: Session ID Prediction Session token analysis and exploitation:

Token Characteristic	Attack Feasibility	Implementation
Sequential numeric (cart_12345)	**Trivial**	Direct iteration
Timestamp-based (1709452800_abc123)	Moderate	Time window brute force
Low-entropy random (8 char alphanumeric)	Moderate	GPU-accelerated cracking
Cryptographically secure (UUID v4)	Infeasible	Abandon, seek alternative vectors

5.2.3 Price Manipulation: ridotto Flag Injection for Adult Tickets Parameter tampering vectors:

Injection Point	Payload	Server-Side Validation Bypass
Direct API parameter	`"price_tier": "ridotto"`	Absent or client-authoritative validation
User profile age modification	`"birth_date": "2005-01-01"`	Age calculation without document verification

Injection Point	Payload	Server-Side Validation Bypass
Group composition claim	`"eu_citizen": true, "age_group": "18-25"`	Self-reported eligibility without verification
Eligibility document bypass	Empty or fake document upload	Automated approval without manual review

5.2.4 Group Reservation Splitting: Large Block Fragmentation Group booking exploitation:

Mechanism	Implementation	Outcome
Multiple parallel reservations	9-person group × 3 sessions = 27 tickets	Circumvent 13-ticket individual limit
Staggered release exploitation	T-7 releases for same group across dates	Consolidate adjacent time slots post-purchase
Cancellation reacquisition	Monitor group cancellations, immediate rebooking	Capture released inventory before public visibility

5.3 Side-Channel Automation

5.3.1 Response Timing Fingerprinting: 200ms vs. 800ms = Available vs. Sold Production timing analysis implementation:

Component	Specification	Purpose
Measurement precision	Microsecond-level (`Instant::now()`)	Detect subtle latency differences
Sample size per condition	100-1000 requests	Statistical significance
Network variance control	Co-located measurement proxy	Isolate server-side latency
Ground truth labeling	Synchronized with known availability states	Supervised learning training data

Machine learning pipeline: feature extraction (mean, variance, percentiles, distribution shape), clas-

sification (random forest, XGBoost, or neural network), and real-time inference with confidence thresholding.

5.3.2 Error Code Oracle: HTTP 410 vs. 404 vs. 500 Semantics **Status code semantic mapping** (empirically validated):

Code	Confirmed Meaning	Strategic Application
200 + "Esaurito"	Sold out, no restock expected	Resource reallocation
200 + "Non disponibile"	Not yet released or invalid	**Release timing prediction, parameter bounds**
302 → waitlist	High demand, restock possible	Sustained monitoring
404	Invalid parameter	**Valid parameter space boundary identification**
410	Previously available, permanently closed	Historical demand analysis
429 + Retry-After: 60	Rate limited, detection triggered	**Evasion tuning feedback, immediate backoff**
500 + stack trace	Server error, potential vulnerability	Information extraction, retry indication

5.3.3 JavaScript Bundle Analysis: Uncovered API Endpoints in Source Maps **Systematic extraction methodology**:

Artifact Location	Extraction Tool	Expected Content
`*.js.map` files	source-map-explorer, unminify	Original TypeScript/React source, internal API documentation
`webpack` runtime	`window.__webpack_require__` inspection	Chunk loading patterns, lazy-loaded endpoints
Service worker (`sw.js`)	Direct fetch and parse	Background sync endpoints, push notification URLs

Artifact Location	Extraction Tool	Expected Content
`window.__CONFIG__`	Browser console extraction	API base URLs, feature flags, environment indicators
GraphQL schema (if present)	Introspection query or schema download	Complete type system, mutation definitions

5.3.4 Mobile App Traffic Mirroring: MyColosseum App API Extraction **Dynamic analysis setup**:

Component	Tool	Configuration
Device/emulator	Android Studio emulator or physical device	Root access for certificate installation
SSL interception	mitmproxy or Burp Suite	Custom CA certificate, SSL pinning bypass
Runtime instrumentation	Frida or objection	Method hooking, parameter logging, memory inspection
Traffic analysis	Wireshark or built-in proxy tools	Protocol identification, endpoint enumeration

6. Payment Automation

6.1 Payment Flow Analysis

6.1.1 Primary Gateway: UniCredit Integration The **UniCredit merchant integration** implements **3D Secure 2.0** with risk-based authentication flow:

Flow Stage	Implementation	Automation Barrier
Card data collection	Hosted fields or redirect	PCI scope avoidance through tokenization
3D Secure initiation	`/3ds/authenticate` endpoint	Challenge presentation determination
Issuer authentication	SMS OTP, banking app, biometric	**Primary automation barrier**
Authorization	`/payment/authorize`	Final confirmation, success detection

Frictionless flow eligibility: low-risk transactions (merchant history, device binding, transaction amount, velocity patterns) may bypass challenge entirely. Optimization target: establish sufficient positive history for automated exemption.

6.1.2 Alternative Methods: PayPal, Bank Transfer (2-3 Day Delay)

Method	Automation Feasibility	Time Constraint	Primary Advantage
PayPal	Moderate	15-minute viable	Pre-authorized consent, established buyer protection
Bank transfer (bonifico)	Very Low	2-3 day processing	Manual verification requirement
Satispay/ Italian wallets	Unknown	Requires investigation	Local payment method diversity

PayPal's **Smart Payment Buttons** and **Checkout API** enable server-side payment creation with customer redirection, potentially automatable with pre-authorized billing agreements.

6.1.3 3D Secure Handling: OTP Interception or Pre-Authorized Cards

Strategy	Implementation	Risk/Legality	Success Rate
Virtual card APIs with 3DS exemption	Stripe Issuing, Privacy.com with negotiated frictionless flow	Contractual, financial compliance	**Highest for legitimate automation**
Banking app notification interception	Android accessibility services, iOS push notification parsing	**Potential CFAA/ computer fraud liability**	Moderate, device-dependent
SMS OTP interception	SIM swapping, SS7 exploitation, forwarding services	**Criminal liability**	High, legally prohibitive
Human-in-the-loop escalation	Real-time notification to operator for challenge completion	Operational overhead, latency	Moderate, scale-limited

6.2 Automation Strategies

6.2.1 Virtual Card APIs: Stripe Issuing, Privacy.com for Per-Ticket Cards

Provider	API Latency	3DS Configuration	Geographic Availability	Cost Structure
Stripe Issuing	<100ms	Programmable with issuer exemption negotiation	EU (limited), US	Per-card + transaction
Privacy.com	200-500ms	US-only, limited 3DS control	US only	Free tier + premium
Revolut Business	300-800ms	Full 3DS with some exemption paths	EU, UK	Subscription + usage
Wise Business	500-1000ms	Standard 3DS, limited customization	Global	Per-transaction

Per-ticket card generation provides: transaction-level reconciliation, immediate cancellation on failure or post-resale, reduced fraud pattern detection through unique card-per-purchase, and geographic BIN matching to proxy location.

6.2.2 Browser Automation Fallback: Playwright for 3D Secure Flows For scenarios requiring interactive 3D Secure completion:

```
// Rust with playwright-rs for 3D Secure handling
use playwright::Playwright;

pub async fn complete_3ds_payment(
    playwright: &Playwright,
    cart_url: &str,
    card_details: &CardDetails,
    otp_receiver: &mut dyn OtpReceiver, // Human or automated
) -> Result<PaymentConfirmation, PaymentError> {
    let browser = playwright.chromium().launch().await?;
    let context = browser.new_context().await?;
    let page = context.new_page().await?;

    // Navigate to cart with session cookies injected
    page.goto(cart_url).await?;

    // Complete checkout flow
    page.fill("[name=cardNumber]", &card_details.number).await?;
    page.fill("[name=expiry]", &card_details.expiry).await?;
    page.fill("[name=cvc]", &card_details.cvc).await?;
    page.click("[type=submit]").await?;

    // Handle 3D Secure challenge
    match page.wait_for_selector(".challenge-iframe", timeout=5000).await {
        Ok(iframe) => {
```

```
            let challenge_frame = iframe.content_frame().await?.unwrap();

            // OTP input required
            let otp = otp_receiver.receive_otp(timeout=120000).await
                .map_err(|_| PaymentError::OtpTimeout)?;

            challenge_frame.fill("[name=otp]", &otp).await?;
            challenge_frame.click("[type=submit]").await?;
        }
        Err(_) => {
            // Frictionless flow - no challenge presented
        }
    }

    // Confirm success
    page.wait_for_selector(".confirmation", timeout=30000).await?;
    let confirmation = extract_confirmation(&page).await?;

    Ok(confirmation)
}
```

6.2.3 Payment Pre-Stage: Tokenized Card-on-File for Sub-Second Completion **Account preparation for minimal latency**:

Preparation Step	Timing	Implementation
Card tokenization	Days before target	Store encrypted card tokens with provider
Billing address verification	Days before target	Pre-verify address matching proxy geography
3D Secure enrollment	Days before target	Complete initial challenge, establish device binding
Risk score optimization	Ongoing	Build positive transaction history, avoid velocity triggers
Payment execution	**<1 second from trigger**	Token retrieval, authorization request, success confirmation

6.2.4 Failure Recovery: Alternative Card Cascade on Decline

Decline Reason	Recovery Strategy	Implementation
Insufficient funds	Immediate secondary card	Pre-staged backup cards with same billing profile
Issuer risk rejection	Geographic BIN rotation	Cards from multiple issuing banks, countries
3D Secure failure	Human escalation or exemption path	Real-time notification, frictionless flow retry
Velocity limit	Card rotation, amount splitting	Multiple smaller transactions across cards
Technical timeout	Idempotent retry with backoff	Exponential backoff, session preservation

6.3 Compliance & Detection Evasion

6.3.1 Velocity Pattern Mimicry: Human-Like Purchase Timing

Automation Behavior	Human Mimicry	Implementation
Immediate purchase on detection	30-120 second deliberation delay	Randomized delay with Pareto distribution (most short, some long)
Constant interaction pace	Variable pace with interruptions	Random pauses, “distraction” periods, return navigation
Perfect form completion	Typing errors, corrections, backtracking	Intentional typo injection with correction
Single-session focus	Multi-tab browsing, comparison shopping	Background tab activity simulation

6.3.2 Device Fingerprint Consistency: Same Session Throughout

Fingerprint Dimension	Consistency Requirement	Violation Detection
TLS/JA3	Identical throughout session	Mid-session change triggers re-authentication

Fingerprint Dimension	Consistency Requirement	Violation Detection
IP address	Stable or plausible mobility pattern	Impossible travel detection
Browser characteristics	No capability changes	Feature detection inconsistency flags
Cookie/session tokens	Progressive refresh, not abrupt replacement	Session hijacking detection
Behavioral biometrics	Consistent mouse/keyboard patterns	Anomaly scoring elevation

6.3.3 Billing Address Rotation: Residential Proxy Geo-Matching

Proxy Location	Required Billing Address	Verification Strategy
Italian residential (Rome)	Rome postal code, Italian name format	Public records correlation
German residential (Munich)	Munich postal code, German name format	Regional phone number format
Mobile carrier IP (Vodafone IT)	Matching carrier billing region	Less strict, carrier-level verification

7. Infrastructure & Deployment

7.1 Personal Infrastructure

7.1.1 Local Hardware: Raspberry Pi 4 Cluster with 4G Failover

Component	Specification	Role	Cost
4× Raspberry Pi 4 8GB	ARM Cortex-A72 1.5GHz	Monitoring workers	~€320
1× Raspberry Pi 4 4GB	ARM Cortex-A72 1.5GHz	Coordination/ notification	~€55
4G LTE modem (Quectel EC25)	150Mbps down, 50Mbps up	Failover connectivity	~€80

Component	Specification	Role	Cost
256GB A2 SD cards	100MB/s read, 40MB/s write	Local persistence	~€120
UPS (APC Back-UPS 650VA)	400W, 5-15 min runtime	Power continuity	~€80

Total: **€655, power consumption** **25W**, capability: **500+ concurrent monitoring streams** with 3-10 second intervals. Suitable for T-7/T-1 acquisition with relaxed latency requirements; T-30 competition demands cloud supplementation.

7.1.2 Residential Proxy Integration: Bright Data, Oxylabs Rotation

Provider	Pool Size	Italian Concentration	Rotation Control	Cost/GB	Recommendation
Bright Data	72M+	15-20% (estimated)	Per-request, session, or sticky	$2-3	**Primary for acquisition**
Oxylabs	100M+	20-25% (estimated)	API-driven, granular targeting	$7-8	Performance-critical paths
Smartproxy	65M+	10-15% (estimated)	Limited rotation modes	$4-5	Cost-optimized backup
PacketStream	P2P residential	Variable	Limited control	$1-2	Experimental, lower trust

Integration: proxy health checking with automatic failover, geographic affinity routing (Italian IPs for Italian targets), and session persistence for cart continuity.

7.1.3 Container Orchestration: Docker Compose with Health Checks

```
# docker-compose.yml for edge deployment

version: '3.8'

services:
  monitor-core:
    image: colosseo-monitor:rust-latest
    deploy:
      replicas: 4
      resources:
        limits:
          cpus: '3.0'
          memory: 2G
```

```
    environment:
      - RUST_LOG=info
      - PROXY_POOL_URL=http://proxy-manager:8080
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:8080/health"]
      interval: 10s
      timeout: 5s
      retries: 3
      start_period: 30s

  orchestrator:
    image: colosseo-orchestrator:go-latest
    ports:
      - "8080:8080"
    environment:
      - TELEGRAM_BOT_TOKEN=${TELEGRAM_TOKEN}
      - REDIS_URL=redis://redis:6379
    depends_on:
      redis:
        condition: service_healthy

  proxy-manager:
    image: goproxy-rotation:latest
    volumes:
      - ./proxy-config:/config
    environment:
      - BRIGHTDATA_USERNAME=${BD_USER}
      - BRIGHTDATA_PASSWORD=${BD_PASS}

  redis:
    image: redis:7-alpine
    volumes:
      - redis-data:/data
    healthcheck:
      test: ["CMD", "redis-cli", "ping"]
      interval: 5s
```

```
volumes:
  redis-data:
```

7.2 Cloud Infrastructure

7.2.1 Provider Selection: Hetzner, OVH for EU Low-Latency

Provider	Location	Instance Type	vCPU/RAM/ Transfer	Monthly Cost	Latency to Rome
Hetzner	Nuremberg, Falkenstein	CPX31 (shared) or CCX13 (dedicated)	4/16GB/20TB or 8/32GB/unlimited	€12.60 or €118.15	~15-25ms
OVH	Gravelines, Strasbourg	VPS Comfort or Advance	4/8GB/unlimited or 8/16GB/unlimited	€11.99 or €31.99	~20-30ms
Scaleway	Paris	DEV1-L or GP1-M	4/8GB/unlimited or 8/16GB/unlimited	€15.99 or €39.99	~25-35ms

Selection criteria: Italian data center preference (none available from major providers), next-best German/French locations with <30ms latency, competitive pricing for sustained workload, and GDPR compliance for EU data residency.

7.2.2 Kubernetes Deployment: Auto-Scaling Monitor Pools

```
# Kubernetes HPA for monitor workers

apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: colosseo-monitor-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: colosseo-monitor
  minReplicas: 3
  maxReplicas: 100
  metrics:

  - type: External

    external:
      metric:
        name: colosseo_target_queue_depth
      target:
        type: AverageValue
```

```
      averageValue: "10"

- type: Resource

  resource:
    name: cpu
    target:
      type: Utilization
      averageUtilization: 70
behavior:
  scaleUp:
    stabilizationWindowSeconds: 0
    policies:

    - type: Percent

      value: 100
      periodSeconds: 15
  scaleDown:
    stabilizationWindowSeconds: 300
    policies:

    - type: Percent

      value: 10
      periodSeconds: 60
```

Pre-positioned scaling: cron-based scale-up 30 minutes before predicted release windows (T-30 at 8:15 AM CET, T-7 variable, T-1 continuous), with rapid scale-down post-release to minimize cost.

7.2.3 Serverless Components: AWS Lambda for Trigger-Only Execution

Function	Trigger	Implementation	Cost Model
Notification dispatcher	SQS message from monitor	Telegram/Discord/webhook multi-channel	Per-invocation, 128MB, <100ms
Payment preparer	Availability detection event	Virtual card token generation, pre-auth	Per-invocation, 256MB, <500ms
Configuration validator	S3 config file change	Schema validation, hot-reload trigger	Per-invocation, 128MB, <200ms
Metrics aggregator	CloudWatch scheduled	Prometheus remote write, dashboard update	Per-invocation, 512MB, <5s

Cold start mitigation: provisioned concurrency for notification dispatcher (critical path), warmed pools for payment preparer during release windows.

7.2.4 Cost Optimization: Spot Instances with Checkpoint Recovery

Workload Type	Instance Class	On-Demand Cost	Spot Cost	Savings	Checkpoint Strategy
Monitor workers	Hetzner CX/ CPX	€0.024/hr	€0.007/hr	**70%**	Stateless, immediate restart
Orchestrator	Hetzner CCX (dedicated)	€0.164/hr	N/A (no spot)	—	Redis-backed, 5s RPO
Payment handler	Reserved instance	€0.089/hr	N/A (latency-critical)	—	Transaction-level idempotency

Checkpoint implementation: Redis persistence for session state, S3/R2 for configuration and logs, graceful termination handling with 30-second drain period.

7.3 Operational Security

7.3.1 Traffic Obfuscation: Domain Fronting via Cloudflare Workers

```
// Cloudflare Worker for domain fronting
export default {
  async fetch(request, env) {
    const url = new URL(request.url);

    // Extract actual target from encrypted header or path
    const targetHeader = request.headers.get('X-Actual-Target');
    const actualTarget = decryptTarget(targetHeader, env.KEY);

    // Rewrite request to actual destination
    const modifiedRequest = new Request(actualTarget, {
      method: request.method,
      headers: filterHeaders(request.headers),
      body: request.body,
    });

    const response = await fetch(modifiedRequest);

    // Strip identifying headers from response
    return new Response(response.body, {
      status: response.status,
      headers: filterResponseHeaders(response.headers),
    });
  }
};
```

Front domain: popular, high-traffic domain with legitimate reason for diverse global access (CDN,

analytics, messaging). **Actual target**: Colosseo ticketing endpoints, distributed across multiple origins for resilience.

7.3.2 Log Sanitization: No Persistent Purchase Records

Log Category	Retention	Sanitization	Purpose
Operational metrics	7 days	Aggregate only, no individual transactions	Performance monitoring, scaling decisions
Error traces	24 hours	Stack trace truncation, no PII	Debugging, rapid response
Security events	30 days	IP anonymization (last octet), no session IDs	Threat detection, pattern analysis
Financial records	Immediate deletion	Never logged to persistent storage	Regulatory compliance, liability avoidance
Successful acquisitions	0 days (ephemeral)	In-memory only, process termination wipe	Operational necessity only, no audit trail

7.3.3 Legal Jurisdiction: Non-EU Control Plane for Regulatory Distance

Component	Jurisdiction	Rationale	Risk Mitigation
Control plane (orchestration, config)	Switzerland, Iceland, or Singapore	Outside EU AGCM direct enforcement	Strong privacy laws, limited cooperation
Worker nodes (monitoring, acquisition)	EU (Germany, France)	Latency optimization, data residency	Minimal persistent state, rapid redeployment
Payment processing	US (Delaware, Wyoming LLCs)	Favorable commercial law, banking access	Corporate veil, limited personal liability
Notification/alerting	Global distributed (Telegram, etc.)	Resilience, jurisdictional diversity	Encrypted, ephemeral messaging

8. Implementation: Core Modules

8.1 Rust Implementation (Performance-Critical Path)

8.1.1 HTTP Client: reqwest with rustls and Custom Cipher Suites

```
// src/client.rs - High-performance HTTP client with fingerprint rotation
use reqwest::{Client, ClientBuilder, header};
use rustls::{ClientConfig, CipherSuite, SupportedCipherSuite};
use std::sync::Arc;
use tokio::sync::RwLock;

pub struct FingerprintedClient {
    inner: Client,
    ja3_rotation: Arc<RwLock<Ja3RotationState>>,
}

#[derive(Clone)]

struct Ja3RotationState {
    cipher_suites: Vec<SupportedCipherSuite>,
    curves: Vec<rustls::NamedGroup>,
    extensions: Vec<u16>,
}

impl FingerprintedClient {
    pub async fn new(proxy_url: Option<&str>) -> Result<Self, Box<dyn std::error::Error>> {
        let state = Self::generate_random_state();

        let tls_config = Self::build_tls_config(&state)?;

        let mut builder = ClientBuilder::new()
            .use_preconfigured_tls(tls_config)
            .http2_prior_knowledge()
            .pool_max_idle_per_host(10)
            .timeout(Duration::from_secs(10))
            .user_agent(Self::random_user_agent());

        if let Some(proxy) = proxy_url {
            builder = builder.proxy(reqwest::Proxy::all(proxy)?);
        }

        let client = builder.build()?;

        Ok(Self {
            inner: client,
            ja3_rotation: Arc::new(RwLock::new(state)),
        })
    }

    fn build_tls_config(state: &Ja3RotationState) -> Result<ClientConfig, rustls::Error> {
        let mut config = ClientConfig::builder()
            .with_safe_defaults()
            .with_root_certificates(Self::root_store())
            .with_no_client_auth();
```

```
    // JA3 fingerprint manipulation through cipher suite ordering
    config.cipher_suites = state.cipher_suites.clone();

    // HTTP/2 and TLS extension configuration
    config.alpn_protocols = vec![b"h2".to_vec(), b"http/1.1".to_vec()];

    Ok(config)
}

fn generate_random_state() -> Ja3RotationState {
    let mut rng = thread_rng();

    // Chrome-like cipher suite selection with random ordering
    let mut ciphers = vec![
        CipherSuite::TLS13_AES_256_GCM_SHA384,
        CipherSuite::TLS13_AES_128_GCM_SHA256,
        CipherSuite::TLS13_CHACHA20_POLY1305_SHA256,
        CipherSuite::TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384,
        CipherSuite::TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384,
    ];
    ciphers.shuffle(&mut rng);

    Ja3RotationState {
        cipher_suites: ciphers.into_iter().map(|c| c.into()).collect(),
        curves: vec![
            rustls::NamedGroup::X25519,
            rustls::NamedGroup::secp256r1,
            rustls::NamedGroup::secp384r1,
        ],
        extensions: vec![0x0000, 0x0017, 0xff01, 0x0005, 0x000a], // SNI,
            extended_master_secret, renegotiation_info, status_request, supported_groups
    }
}

fn random_user_agent() -> &'static str {
    const UAS: &[&str] = &[
        "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
            Chrome/120.0.0.0 Safari/537.0",
        "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like Gecko
            ) Chrome/120.0.0.0 Safari/537.0",
        "Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome
            /120.0.0.0 Safari/537.0",
    ];
    UAS[thread_rng().gen_range(0..UAS.len())]
}

pub async fn rotate_fingerprint(&self) {
    let mut state = self.ja3_rotation.write().await;

    *state = Self::generate_random_state();
```

```
        // Note: Requires client reconstruction for TLS config changes
        // In production, maintain client pool with pre-rotated instances
    }
}
```

8.1.2 HTML Parsing: scraper with Selector-Based Extraction

```
// src/parser.rs - High-performance HTML availability extraction
use scraper::{Html, Selector, ElementRef};
use std::collections::HashMap;

pub struct AvailabilityParser {
    // Multiple selector strategies for resilience against UI changes
    selectors: Vec<(Selector, AvailabilityIndicator)>,
}

#[derive(Clone, Debug)]

pub enum AvailabilityIndicator {
    Available, // Green highlight, "Disponibile", clickable time slot
    SoldOut, // "Esaurito", grayed out, disabled button
    NotYetReleased, // "Non disponibile" without date-specific messaging
    Loading, // Spinner, skeleton UI, indeterminate state
}

impl AvailabilityParser {
    pub fn new() -> Result<Self, Box<dyn std::error::Error>> {
        let selectors = vec![
            // Primary: calendar day availability
            (Selector::parse("div.calendar-day.available")?, AvailabilityIndicator::Available),
            (Selector::parse("div.calendar-day.sold-out")?, AvailabilityIndicator::SoldOut),

            // Secondary: time slot buttons
            (Selector::parse("button.time-slot:not([disabled])")?, AvailabilityIndicator::
                Available),
            (Selector::parse("button.time-slot[disabled]")?, AvailabilityIndicator::SoldOut),

            // Tertiary: text-based indicators
            (Selector::parse("span.disponibile")?, AvailabilityIndicator::Available),
            (Selector::parse("span.esaurito")?, AvailabilityIndicator::SoldOut),

            // Quaternary: data attributes
            (Selector::parse("[data-availability='true']")?, AvailabilityIndicator::Available),
            (Selector::parse("[data-availability='false']")?, AvailabilityIndicator::SoldOut),
        ];

        Ok(Self { selectors })
    }

    pub fn parse(&self, html: &str) -> ParsedAvailability {
```

```
        let document = Html::parse_document(html);
        let mut results = HashMap::new();

        for (selector, indicator) in &self.selectors {
            for element in document.select(selector) {
                let key = self.extract_key(&element);
                results.entry(key).or_insert_with(Vec::new).push(indicator.clone());
            }
        }

        // Consensus resolution: require multiple indicators for confidence
        ParsedAvailability {
            slots: results.into_iter()
                .map(|(k, v)| (k, self.resolve_consensus(v)))
                .collect(),
            confidence: self.calculate_confidence(&results),
            raw_indicators: results.len(),
        }
    }

    fn extract_key(&self, element: &ElementRef) -> String {
        // Extract date/time identifier from element context
        element.value().attr("data-date")
            .or_else(|| element.value().attr("data-time"))
            .or_else(|| element.parent().and_then(|p| p.value().attr("data-date")))
            .map(|s| s.to_string())
            .unwrap_or_else(|| "unknown".to_string())
    }

    fn resolve_consensus(&self, indicators: Vec<AvailabilityIndicator>) -> AvailabilityStatus {
        // Weight by selector reliability, require agreement for high confidence
        let available_count = indicators.iter().filter(|i| matches!(i, AvailabilityIndicator::
            Available)).count();
        let sold_out_count = indicators.iter().filter(|i| matches!(i, AvailabilityIndicator::
            SoldOut)).count();

        match (available_count, sold_out_count) {
            (a, s) if a > s && a >= 2 => AvailabilityStatus::Available(Confidence::High),
            (a, s) if s > a && s >= 2 => AvailabilityStatus::SoldOut(Confidence::High),
            (a, s) if a > 0 || s > 0 => AvailabilityStatus::Uncertain,
            _ => AvailabilityStatus::NoData,
        }
    }
}
```

8.1.3 Async Runtime: tokio with work-stealing Scheduler

```
// src/main.rs - Tokio runtime configuration for monitoring
use tokio::runtime::{Builder, Runtime};
use std::num::NonZeroUsize;
```

```
pub fn create_optimized_runtime() -> Runtime {
    Builder::new_multi_thread()
        .worker_threads(num_cpus::get())
        .max_blocking_threads(512)
        .thread_stack_size(2 * 1024 * 1024) // 2MB stack for deep call chains
        .enable_all()
        .event_interval(61) // Optimal for high-frequency I/O
        .global_queue_interval(61)
        .max_io_events_per_tick(1024)
        .build()
        .expect("Failed to create tokio runtime")
}

// Per-target monitoring task with priority-aware scheduling
pub async fn spawn_monitors(
    runtime: &Runtime,
    targets: Vec<MonitorTarget>,
    priority: TaskPriority,
) -> Vec<JoinHandle<()>> {
    let semaphore = Arc::new(Semaphore::new(match priority {
        TaskPriority::Critical => 100, // Maximum concurrency for T-30
        TaskPriority::High => 50, // T-7 windows
        TaskPriority::Normal => 20, // Background monitoring
    }));

    targets.into_iter().map(|target| {
        let sem = semaphore.clone();
        runtime.spawn(async move {
            let _permit = sem.acquire().await.unwrap();
            run_monitor(target).await
        })
    }).collect()
}
```

8.1.4 State Machine: ticket_states for Lifecycle Management

```
// src/state_machine.rs - Explicit state management for acquisition lifecycle
use std::time::{Duration, Instant};
use serde::{Serialize, Deserialize};

#[derive(Clone, Debug, Serialize, Deserialize)]

pub enum TicketState {
    // Monitoring phase
    Monitoring {
        started_at: Instant,
        last_check: Instant,
        check_count: u64,
    },

    // Detection phase
```

```
    Detected {
        detected_at: Instant,
        confidence: f32,
        signals: Vec<DetectionSignal>,
    },

    // Acquisition phase
    Carting {
        started_at: Instant,
        session_id: String,
        cart_id: Option<String>,
    },

    Holding {
        cart_id: String,
        expires_at: Instant,
        heartbeat_due: Instant,
    },

    // Payment phase
    Paying {
        started_at: Instant,
        payment_token: String,
        method: PaymentMethod,
    },

    // Terminal states
    Confirmed {
        confirmed_at: Instant,
        confirmation_code: String,
        tickets: Vec<TicketDetail>,
    },

    Failed {
        failed_at: Instant,
        reason: FailureReason,
        retry_eligible: bool,
    },
}

impl TicketState {
    pub fn can_transition(&self, to: &TicketState) -> bool {
        use TicketState::*;
        matches!((self, to),
            (Monitoring{..}, Detected{..}) |
            (Detected{..}, Carting{..}) |
            (Carting{..}, Holding{..}) |
            (Carting{..}, Failed{..}) |
            (Holding{..}, Paying{..}) |
            (Holding{..}, Failed{..}) |
            (Paying{..}, Confirmed{..}) |
```

```
            (Paying{..}, Failed{..}) |
            (Failed{retry_eligible: true, ..}, Monitoring{..}) |
            (Failed{retry_eligible: true, ..}, Carting{..})
        )
    }

    pub fn timeout(&self) -> Option<Duration> {
        use TicketState::*;
        match self {
            Monitoring{..} => Some(Duration::from_secs(86400)), // 24h max monitoring
            Detected{..} => Some(Duration::from_secs(5)), // 5s to initiate carting
            Carting{..} => Some(Duration::from_secs(10)), // 10s to secure hold
            Holding{expires_at, ..} => Some(*expires_at - Instant::now()),
            Paying{..} => Some(Duration::from_secs(60)), // 60s for payment completion
            _ => None,
        }
    }
}
```

8.2 Go Implementation (Rapid Deployment)

8.2.1 Scraping Engine: colly with Distributed Collector

```
// cmd/orchestrator/main.go - Colly-based monitoring with distributed coordination
package main

import (
    "context"
    "fmt"
    "log"
    "net/http"
    "os"
    "os/signal"
    "syscall"
    "time"

    "github.com/gocolly/colly/v2"
    "github.com/gocolly/colly/v2/extensions"
    "github.com/gocolly/colly/v2/proxy"
    "github.com/gocolly/colly/v2/storage"
    "github.com/spf13/viper"
)

type MonitorConfig struct {
    Targets []Target `mapstructure:"targets"`
    ProxyPool ProxyConfig `mapstructure:"proxy_pool"`
    Telegram TelegramConfig `mapstructure:"telegram"`
    PollInterval time.Duration `mapstructure:"poll_interval"`
    MaxDepth int `mapstructure:"max_depth"`
    AsyncThreads int `mapstructure:"async_threads"`
}
```

```
type Target struct {
    Name string `mapstructure:"name"`
    URL string `mapstructure:"url"`
    TicketType string `mapstructure:"ticket_type"`
    Selectors map[string]string `mapstructure:"selectors"`
    Headers map[string]string `mapstructure:"headers"`
    Priority int `mapstructure:"priority"`
    Timeout time.Duration `mapstructure:"timeout"`
}

func main() {
    ctx, cancel := context.WithCancel(context.Background())
    defer cancel()

    // Configuration loading with hot-reload
    viper.SetConfigName("config")
    viper.SetConfigType("yaml")
    viper.AddConfigPath(".")
    viper.AddConfigPath("/etc/colosseo/")

    if err := viper.ReadInConfig(); err != nil {
        log.Fatalf("Config␣error:␣%v", err)
    }

    var cfg MonitorConfig
    if err := viper.Unmarshal(&cfg); err != nil {
        log.Fatalf("Config␣unmarshal␣error:␣%v", err)
    }

    // Hot reload on config change
    viper.OnConfigChange(func(e fsnotify.Event) {
        log.Printf("Config␣changed:␣%s", e.Name)
        var newCfg MonitorConfig
        if err := viper.Unmarshal(&newCfg); err != nil {
            log.Printf("Failed␣to␣reload␣config:␣%v", err)
            return
        }
        // Apply new configuration atomically
        updateConfig(&cfg, &newCfg)
    })
    viper.WatchConfig()

    // Initialize distributed components
    redisClient := initRedis(cfg.Redis)
    telegramBot := initTelegram(cfg.Telegram)
    metrics := initPrometheus()

    // Create collectors per target with shared proxy pool
    collectors := make(map[string]*colly.Collector)
    for _, target := range cfg.Targets {
```

```
        collectors[target.Name] = createCollector(target, cfg, redisClient)
    }

    // Start HTTP API for external control
    go startAPI(ctx, &cfg, collectors, redisClient)

    // Start monitoring loops
    var wg sync.WaitGroup
    for name, collector := range collectors {
        wg.Add(1)
        go runMonitor(ctx, &wg, name, collector, cfg.Targets[name], telegramBot, metrics)
    }

    // Graceful shutdown
    sigChan := make(chan os.Signal, 1)
    signal.Notify(sigChan, syscall.SIGINT, syscall.SIGTERM)
    <-sigChan

    log.Println("Shutting␣down...")
    cancel()
    wg.Wait()
}

func createCollector(target Target, cfg MonitorConfig, redis *redis.Client) *colly.Collector {
    c := colly.NewCollector(
        colly.UserAgent(randomUserAgent()),
        colly.AllowedDomains("ticketing.colosseo.it", "www.colosseo.it"),
        colly.MaxDepth(cfg.MaxDepth),
        colly.Async(true),
    )

    // Storage for session persistence
    c.SetStorage(&storage.RedisStorage{
        Address: redis.Options().Addr,
        Password: redis.Options().Password,
        DB: redis.Options().DB,
        Prefix: fmt.Sprintf("colly:%s:", target.Name),
    })

    // Proxy rotation with health checking
    proxyFunc := createProxyPool(cfg.ProxyPool)
    c.SetProxyFunc(proxyFunc)

    // Rate limiting with adaptive jitter
    c.Limit(&colly.LimitRule{
        DomainGlob: "*colosseo.it*",
        Parallelism: cfg.AsyncThreads,
        Delay: cfg.PollInterval,
        RandomDelay: cfg.PollInterval / 2,
    })
```

```
    // Extensions
    extensions.RandomUserAgent(c)
    extensions.Referer(c)

    // Custom headers from config
    for k, v := range target.Headers {
        c.OnRequest(func(r *colly.Request) {
            r.Headers.Set(k, v)
        })
    }

    // Callbacks
    c.OnHTML(target.Selectors["available"], func(e *colly.HTMLElement) {
        handleAvailability(e, target, true)
    })
    c.OnHTML(target.Selectors["sold_out"], func(e *colly.HTMLElement) {
        handleAvailability(e, target, false)
    })
    c.OnError(func(r *colly.Response, err error) {
        handleError(r, err, target)
    })

    return c
}

func runMonitor(ctx context.Context, wg *sync.WaitGroup, name string, c *colly.Collector, target
     Target, bot *tgbotapi.BotAPI, metrics *prometheus.Registry) {
    defer wg.Done()

    ticker := time.NewTicker(cfg.PollInterval)
    defer ticker.Stop()

    for {
        select {
        case <-ctx.Done():
            return
        case <-ticker.C:
            if err := c.Visit(target.URL); err != nil {
                metrics.Counter("visit_errors").Inc()
                log.Printf("[%s] Visit error: %v", name, err)
            }
            c.Wait() // Wait for async completion
        }
    }
}
```

8.2.2 Proxy Management: goproxy with Live Health Checking

```
// internal/proxy/manager.go - Dynamic proxy pool with health monitoring
package proxy
```

```
import (
    "context"
    "net/http"
    "net/url"
    "sync"
    "time"

    "github.com/prometheus/client_golang/prometheus"
)

type Proxy struct {
    URL *url.URL
    HealthScore float64 // 0-1, based on success rate and latency
    LastUsed time.Time
    LastError error
    ConsecutiveErrors int
    BannedUntil time.Time
    Geographic string // "IT", "DE", "FR", etc.
    ASN string // ISP identifier
}

type Manager struct {
    proxies []*Proxy
    mu sync.RWMutex
    healthCheckInterval time.Duration
    metrics *prometheus.CounterVec
}

func NewManager(proxyURLs []string, checkInterval time.Duration) (*Manager, error) {
    m := &Manager{
        proxies: make([]*Proxy, 0, len(proxyURLs)),
        healthCheckInterval: checkInterval,
        metrics: prometheus.NewCounterVec(prometheus.CounterOpts{
            Name: "proxy_requests_total",
            Help: "Total␣requests␣by␣proxy␣and␣status",
        }, []string{"proxy", "status"}),
    }

    for _, u := range proxyURLs {
        parsed, err := url.Parse(u)
        if err != nil {
            return nil, fmt.Errorf("invalid␣proxy␣URL␣%s:␣%w", u, err)
        }
        m.proxies = append(m.proxies, &Proxy{
            URL: parsed,
            HealthScore: 1.0,
            Geographic: extractGeographic(parsed),
        })
    }

    go m.healthCheckLoop()
```

```
    return m, nil
}

func (m *Manager) GetProxy(preferredGeo string) *url.URL {
    m.mu.RLock()
    defer m.mu.RUnlock()

    // Filter healthy, non-banned proxies
    candidates := make([]*Proxy, 0)
    for _, p := range m.proxies {
        if p.BannedUntil.After(time.Now()) {
            continue
        }
        if p.HealthScore < 0.3 {
            continue
        }
        candidates = append(candidates, p)
    }

    if len(candidates) == 0 {
        // Fallback: least recently used, regardless of health
        return m.fallbackProxy()
    }

    // Weighted selection by health score and geographic preference
    var totalWeight float64
    for _, p := range candidates {
        weight := p.HealthScore
        if p.Geographic == preferredGeo {
            weight *= 2.0 // Geographic preference bonus
        }
        totalWeight += weight
    }

    r := rand.Float64() * totalWeight
    for _, p := range candidates {
        weight := p.HealthScore
        if p.Geographic == preferredGeo {
            weight *= 2.0
        }
        r -= weight
        if r <= 0 {
            p.LastUsed = time.Now()
            return p.URL
        }
    }

    return candidates[0].URL
}

func (m *Manager) ReportResult(proxyURL *url.URL, success bool, latency time.Duration) {
```

```
    m.mu.Lock()
    defer m.mu.Unlock()

    for _, p := range m.proxies {
        if p.URL.String() == proxyURL.String() {
            if success {
                p.ConsecutiveErrors = 0
                p.HealthScore = min(1.0, p.HealthScore*1.1+0.05)
                m.metrics.WithLabelValues(p.URL.Host, "success").Inc()
            } else {
                p.ConsecutiveErrors++
                p.HealthScore *= 0.8
                if p.ConsecutiveErrors > 5 {
                    p.BannedUntil = time.Now().Add(time.Duration(p.ConsecutiveErrors) * time.Minute
                        )
                }
                m.metrics.WithLabelValues(p.URL.Host, "error").Inc()
            }
            break
        }
    }
}

func (m *Manager) healthCheckLoop() {
    ticker := time.NewTicker(m.healthCheckInterval)
    defer ticker.Stop()

    for range ticker.C {
        m.runHealthChecks()
    }
}

func (m *Manager) runHealthChecks() {
    var wg sync.WaitGroup
    for _, p := range m.proxies {
        wg.Add(1)
        go func(proxy *Proxy) {
            defer wg.Done()

            client := &http.Client{
                Timeout: 10 * time.Second,
                Transport: &http.Transport{
                    Proxy: http.ProxyURL(proxy.URL),
                },
            }

            start := time.Now()
            resp, err := client.Get("https://ticketing.colosseo.it/health") // Or equivalent
                lightweight endpoint
            latency := time.Since(start)
```

```
            success := err == nil && resp.StatusCode == 200
            m.ReportResult(proxy.URL, success, latency)
        }(p)
    }
    wg.Wait()
}
```

8.2.3 Notification Pipeline: WebSocket + Telegram Bot API

```
// internal/notify/dispatcher.go - Multi-channel notification with fallback
package notify

import (
    "context"
    "fmt"
    "time"

    tgbotapi "github.com/go-telegram-bot-api/telegram-bot-api/v5"
    "github.com/gorilla/websocket"
)

type Dispatcher struct {
    telegram *tgbotapi.BotAPI
    chatID int64
    webSocket *websocket.Conn
    webhookURL string
    fallbackChan chan<- Alert
}

type Alert struct {
    Level AlertLevel // Info, Warning, Critical
    Timestamp time.Time
    Target string
    Availability AvailabilityStatus
    Confidence float32
    Screenshot []byte // Optional visual confirmation
    Metadata map[string]interface{}
}

func (d *Dispatcher) Dispatch(ctx context.Context, alert Alert) error {
    var errs []error

    // Primary: Telegram for critical alerts
    if alert.Level >= Warning {
        if err := d.sendTelegram(alert); err != nil {
            errs = append(errs, fmt.Errorf("telegram: %w", err))
        }
    }

    // Secondary: WebSocket for real-time dashboard
    if d.webSocket != nil {
```

```
        if err := d.sendWebSocket(alert); err != nil {
            errs = append(errs, fmt.Errorf("websocket: %w", err))
        }
    }

    // Tertiary: Webhook for external integration
    if d.webhookURL != "" {
        if err := d.sendWebhook(alert); err != nil {
            errs = append(errs, fmt.Errorf("webhook: %w", err))
        }
    }

    // Fallback: channel-based for internal handling
    if len(errs) > 0 && d.fallbackChan != nil {
        select {
        case d.fallbackChan <- alert:
        default: // Non-blocking
        }
    }

    if len(errs) == 3 { // All failed
        return fmt.Errorf("all notification channels failed: %v", errs)
    }
    return nil
}

func (d *Dispatcher) sendTelegram(alert Alert) error {
    var msg string
    switch alert.Level {
    case Critical:
        msg = fmt.Sprintf("🚨 *CRITICAL: Tickets Available*\n\n"+
            "Target: %s\n"+
            "Time: %s\n"+
            "Confidence: %.0f%%\n"+
            "Status: %s",
            alert.Target,
            alert.Timestamp.Format("15:04:05.000"),
            alert.Confidence*100,
            alert.Availability,
        )
    case Warning:
        msg = fmt.Sprintf("⚠️ *WARNING: Possible Availability*\n\n"+
            "Target: %s\n"+
            "Confidence: %.0f%%",
            alert.Target,
            alert.Confidence*100,
        )
    default:
        msg = fmt.Sprintf("ℹ️ Info: %s - %s", alert.Target, alert.Availability)
    }
```

```
    tgMsg := tgbotapi.NewMessage(d.chatID, msg)
    tgMsg.ParseMode = "Markdown"
    tgMsg.DisableWebPagePreview = true

    // Include screenshot if available and critical
    if alert.Level == Critical && len(alert.Screenshot) > 0 {
        photo := tgbotapi.NewPhoto(d.chatID, tgbotapi.FileBytes{
            Name: "confirmation.png",
            Bytes: alert.Screenshot,
        })
        photo.Caption = msg
        photo.ParseMode = "Markdown"
        _, err := d.telegram.Send(photo)
        return err
    }

    _, err := d.telegram.Send(tgMsg)
    return err
}
```

8.2.4 Configuration: Viper for Hot-Reloading Targets

```
// internal/config/manager.go - Dynamic configuration with validation
package config

import (
    "fmt"
    "sync"

    "github.com/fsnotify/fsnotify"
    "github.com/spf13/viper"
)

type Manager struct {
    viper *viper.Viper
    current *Config
    mu sync.RWMutex
    watchers []func(*Config)
}

type Config struct {
    Version int `mapstructure:"version"`
    Targets []Target `mapstructure:"targets"`
    UpdatedAt time.Time `mapstructure:"-"`
}

func NewManager(configPath string) (*Manager, error) {
    v := viper.New()
    v.SetConfigFile(configPath)
    v.SetConfigType("yaml")
```

```
    // Defaults
    v.SetDefault("poll_interval", 5*time.Second)
    v.SetDefault("max_depth", 2)
    v.SetDefault("async_threads", 4)

    if err := v.ReadInConfig(); err != nil {
        return nil, fmt.Errorf("read␣config:␣%w", err)
    }

    m := &Manager{
        viper: v,
    }

    if err := m.load(); err != nil {
        return nil, err
    }

    v.WatchConfig()
    v.OnConfigChange(func(e fsnotify.Event) {
        if err := m.load(); err != nil {
            log.Printf("Config␣reload␣failed:␣%v", err)
            return
        }
        m.notifyWatchers()
    })

    return m, nil
}

func (m *Manager) load() error {
    var cfg Config
    if err := m.viper.Unmarshal(&cfg); err != nil {
        return fmt.Errorf("unmarshal:␣%w", err)
    }

    // Validation
    if err := validate(&cfg); err != nil {
        return fmt.Errorf("validation:␣%w", err)
    }

    cfg.UpdatedAt = time.Now()

    m.mu.Lock()
    m.current = &cfg
    m.mu.Unlock()

    return nil
}

func (m *Manager) Get() *Config {
    m.mu.RLock()
```

```
    defer m.mu.RUnlock()
    return m.current
}

func (m *Manager) OnChange(fn func(*Config)) {
    m.watchers = append(m.watchers, fn)
}

func (m *Manager) notifyWatchers() {
    cfg := m.Get()
    for _, fn := range m.watchers {
        go fn(cfg) // Async notification
    }
}

func validate(cfg *Config) error {
    if len(cfg.Targets) == 0 {
        return fmt.Errorf("no␣targets␣configured")
    }
    for i, t := range cfg.Targets {
        if t.URL == "" {
            return fmt.Errorf("target␣%d:␣missing␣URL", i)
        }
        if t.Name == "" {
            return fmt.Errorf("target␣%d:␣missing␣name", i)
        }
    }
    return nil
}
```

8.3 Hybrid Coordination Layer

8.3.1 gRPC Service Mesh: Rust Workers, Go Orchestrators

```
// proto/colosseo.proto
syntax = "proto3";
package colosseo;

service Monitor {
  rpc StartMonitoring(MonitorRequest) returns (stream AvailabilityEvent);
  rpc StopMonitoring(StopRequest) returns (StopResponse);
  rpc GetStatus(StatusRequest) returns (StatusResponse);
}

service Acquisition {
  rpc InitiateAcquisition(AcquisitionRequest) returns (AcquisitionResponse);
  rpc GetCartStatus(CartStatusRequest) returns (CartStatusResponse);
  rpc CompletePayment(PaymentRequest) returns (PaymentResponse);
}

message MonitorRequest {
```

```
  string target_id = 1;
  string url = 2;
  TicketType ticket_type = 3;
  MonitoringConfig config = 4;
}

message AvailabilityEvent {
  string target_id = 1;
  int64 timestamp_ms = 2;
  AvailabilityStatus status = 3;
  float confidence = 4;
  repeated DetectionSignal signals = 5;
  map<string, string> metadata = 6;
}

message AcquisitionRequest {
  string target_id = 1;
  string event_id = 2;
  int32 quantity = 3;
  PriceTier tier = 4;
  repeated Visitor visitors = 5;
  PaymentMethod preferred_payment = 6;
}

message AcquisitionResponse {
  string cart_id = 1;
  int64 hold_expires_ms = 2;
  repeated PaymentOption payment_options = 3;
}

enum TicketType {
  UNKNOWN = 0;
  ORDINARIO = 1;
  FULL_EXPERIENCE_ARENA = 2;
  FULL_EXPERIENCE_UNDERGROUND = 3;
  FORUM_PASS_SUPER = 4;
}

enum AvailabilityStatus {
  UNKNOWN_STATUS = 0;
  AVAILABLE = 1;
  SOLD_OUT = 2;
  NOT_YET_RELEASED = 3;
  UNCERTAIN = 4;
}
```

8.3.2 Redis State Store: Distributed Locks for Cart Claims

```
// src/redis.rs - Distributed coordination with Redis
use redis::{aio::MultiplexedConnection, AsyncCommands, RedisResult};
use std::time::Duration;
```

```rust
pub struct StateStore {
    conn: MultiplexedConnection,
    key_prefix: String,
}

impl StateStore {
    pub async fn new(redis_url: &str) -> Result<Self, redis::RedisError> {
        let client = redis::Client::open(redis_url)?;
        let conn = client.get_multiplexed_async_connection().await?;
        Ok(Self {
            conn,
            key_prefix: "colosseo:".to_string(),
        })
    }

    // Distributed lock for cart claim coordination
    pub async fn try_claim_cart(&self, event_id: &str, cart_id: &str, ttl_secs: u64) ->
         RedisResult<bool> {
        let key = format!("{}cart_claim:{}", self.key_prefix, event_id);
        let value = cart_id;

        // NX: only set if not exists; EX: expire after ttl_secs
        let result: Option<String> = self.conn.set_nx_ex(&key, value, ttl_secs).await?;
        Ok(result.is_some())
    }

    // Release claim (on successful purchase or abandonment)
    pub async fn release_claim(&self, event_id: &str) -> RedisResult<()> {
        let key = format!("{}cart_claim:{}", self.key_prefix, event_id);
        self.conn.del(&key).await
    }

    // State machine persistence
    pub async fn save_state(&self, target_id: &str, state: &TicketState) -> RedisResult<()> {
        let key = format!("{}state:{}", self.key_prefix, target_id);
        let value = serde_json::to_string(state).map_err(|e| {
            redis::RedisError::from((redis::ErrorKind::TypeError, "Serialization failed", e.
                to_string()))
        })?;
        self.conn.set_ex(&key, value, 3600).await // 1 hour TTL
    }

    pub async fn load_state(&self, target_id: &str) -> RedisResult<Option<TicketState>> {
        let key = format!("{}state:{}", self.key_prefix, target_id);
        let value: Option<String> = self.conn.get(&key).await?;
        match value {
            Some(v) => serde_json::from_str(&v).map_err(|e| {
                redis::RedisError::from((redis::ErrorKind::TypeError, "Deserialization failed", e.
                    to_string()))
            }).map(Some),
```

```
            None => Ok(None),
        }
    }

    // Metrics aggregation
    pub async fn record_metric(&self, name: &str, value: f64, labels: &[(&str, &str)]) ->
        RedisResult<()> {
        let key = format!("{}metric:{}:{}", self.key_prefix, name,
            labels.iter().map(|(k,v)| format!("{}={}", k, v)).collect::<Vec<_>>().join(","));
        let timestamp = chrono::Utc::now().timestamp_millis();

        // Time-series: sorted set by timestamp
        self.conn.zadd(&key, timestamp, value.to_string()).await?;
        // Retention: trim to last 24 hours
        self.conn.zremrangebyscore(&key, 0, timestamp - 86400000).await?;
        Ok(())
    }
}
```

8.3.3 Prometheus Metrics: Success Rate, Latency, Ban Detection

```
# prometheus/colosseo.yml - Custom metrics configuration

groups:

  - name: colosseo_monitoring

    interval: 15s
    rules:

      - record: colosseo:availability_detection_rate

        expr: |
          sum(rate(colosseo_availability_events_total{status="detected"}[5m]))
          /
          sum(rate(colosseo_poll_attempts_total[5m]))

      - record: colosseo:acquisition_success_rate

        expr: |
          sum(rate(colosseo_acquisitions_total{status="confirmed"}[1h]))
          /
          sum(rate(colosseo_acquisitions_total[1h]))

      - record: colosseo:proxy_ban_rate

        expr: |
          sum(rate(colosseo_proxy_errors_total{reason="banned"}[10m]))
          /
          sum(rate(colosseo_proxy_requests_total[10m]))
```

```
- alert: HighProxyBanRate

  expr: colosseo:proxy_ban_rate > 0.1
  for: 5m
  labels:
    severity: warning
  annotations:
    summary: "Proxy ban rate elevated: {{ $value | humanizePercentage }}"

- alert: AcquisitionSuccessRateDrop

  expr: colosseo:acquisition_success_rate < 0.05
  for: 15m
  labels:
    severity: critical
  annotations:
    summary: "Acquisition success rate critically low"
```

9. Target Ticket Type Specialization

9.1 Full Experience Arena (Highest Value)

9.1.1 Monitoring Priority: Maximum Polling Frequency The **Full Experience Arena** ticket category represents the optimal risk-adjusted return for automation investment. With **€24 adult pricing** and **reconstructed arena floor access** creating substantial visitor value, secondary market premiums of 50-200% are achievable during peak demand periods. The category's **moderate scarcity**—more available than underground access but significantly constrained versus standard entry—creates competitive dynamics where sophisticated automation achieves meaningful market share.

Resource allocation: Dedicated worker pool with **500ms effective polling interval** (distributed across 10-20 sessions to avoid per-session rate limits), Italian residential proxy concentration for minimal latency, and **pre-staged payment infrastructure** with sub-5-second completion capability.

9.1.2 Release Pattern: T-30 Bulk, Scattered T-7/T-1

Release Window	Typical Availability	Optimal Strategy
T-30 (8:45 CET)	50-200 tickets/day, 5-15 per slot	Maximum infrastructure, parallel session saturation
T-7 (variable)	20-50 tickets/day, tour operator returns	Sustained monitoring, reduced competition
T-1 (continuous)	10-30 tickets, cancellation-driven	Aggressive polling with immediate human escalation

9.1.3 Competition Analysis: Tour Operator Bot Detection Post-2025 regulatory enforcement, **tour operator automation has been substantially curtailed** through reduced inventory allocation

(25% versus historical 40-50%) and enhanced B2B monitoring. This competitive reduction creates **arbitrage opportunity for individual automation operators** with sufficient technical sophistication to evade consumer-facing defenses. Remaining competition includes: individual scalpers with primitive automation (easily outperformed), international visitors with manual refresh (time zone disadvantaged), and residual tour operator systems (reduced scale, detectable patterns).

9.2 Ordinario (Volume Play)

9.2.1 Sustained Availability: Lower Competition, Predictable Restock **Standard entry tickets** (€18 adult) maintain **hours-to-days availability post-release**, enabling volume-focused strategies rather than speed-critical acquisition. The lower per-ticket margin is offset by: higher acquisition success rate (>80% versus <20% for Full Experience), group purchase optimization (13-ticket maximum exploitation), and upgrade path monetization (on-site Full Experience add-on availability).

9.2.2 Group Purchase Optimization: 13-Ticket Maximum Exploitation The **13-ticket per-transaction limit**creates optimization requirements for larger groups: parallel session coordination for simultaneous purchases, staggered timing to avoid velocity detection, and post-purchase consolidation if adjacent time slots are acceptable.

9.2.3 Upgrade Path: On-Site Full Experience Add-On **Revenue enhancement** through on-site upgrade availability monitoring: visitors with standard entry tickets may purchase arena or underground access upgrades at venue kiosks, with availability distinct from advance purchase inventory. Automation extension to monitor upgrade availability creates secondary monetization opportunity.

9.3 Forum Pass Super (Niche Arbitrage)

9.3.1 Mispricing Opportunities: Bundle Component Valuation The **Forum Pass Super** (€18) provides access to Roman Forum, Palatine Hill, and SUPER sites **without Colosseum entry**. Bundle component analysis reveals potential mispricing: individual site access sums exceed pass price, creating value for visitors with alternative Colosseum access (guided tours, Roma Pass, separate purchase). Automation targeting identifies **price-sensitive segments** and **bundle optimization opportunities**.

9.3.2 Cross-Site Comparison: GetYourGuide, Tiqets Price Deltas **Platform arbitrage** monitoring: third-party resellers (GetYourGuide, Tiqets, Musement) frequently price identical inventory with 10-30% premiums or discounts based on their demand forecasting and customer segmentation. Automated price comparison across platforms identifies **buy-low-sell-high opportunities** for inventory acquired at official prices.

10. Detection Evasion & Countermeasures

10.1 Behavioral Mimicry

10.1.1 Mouse Movement Simulation: Bezier Curves for Cursor Paths Human mouse movement exhibits **subtle curvature and velocity variation** absent from linear automation. Implementation: cubic Bezier curves between start and end points with control point randomization, velocity profiles with acceleration/deceleration (not constant speed), and occasional overshoot with correction.

10.1.2 Scroll Pattern Randomization: Variable Velocity Profiles Scroll behavior: **non-uniform velocity** (fast initial, deceleration near content of interest), **direction changes** (upward scrolls to re-read, hesitation), and **pause patterns** (reading simulation, not continuous movement).

10.1.3 Form Interaction Delays: Human Typing Simulation Text entry: **inter-key timing variation** (80-200ms with occasional 500ms+ pauses), **typo injection with correction** (3-5% error rate), and **field navigation** (Tab versus click, with variation).

10.2 Technical Fingerprint Evasion

10.2.1 TLS/JA3 Fingerprint Rotation: Custom Client Hello Per-session JA3 hash variation through: cipher suite subset selection (16+ possible combinations from 5+ supported), extension ordering permutation, and elliptic curve preference shuffling. The `utls` library (Go) or `rustls` with custom `ClientConfig` enables programmatic generation of **unique but plausible browser fingerprints**.

10.2.2 HTTP/2 Settings Randomization: Window Size, Header Table HTTP/2 SETTINGS frame parameters: initial window size (65,535 to 16,777,215), header table size (4,096 to 65,536), and maximum concurrent streams (100 to 1,000) varied per connection to defeat fingerprint databases.

10.2.3 WebGL/Canvas Noise: Unique Per-Session Rendering Canvas and WebGL fingerprint randomization through: subtle pixel-level noise injection (imperceptible to human, detectable by hash), font rendering variation, and hardware capability reporting modification.

10.3 Infrastructure-Level Evasion

10.3.1 Residential IP Rotation: ASN Diversity Requirements Minimum viable proxy diversity: **5+ distinct ASNs** (Telecom Italia, Vodafone, WindTre, Fastweb, Tiscali), **3+ geographic regions within Italy** (Rome, Milan, Naples), and **mobile carrier inclusion** (TIM, Vodafone, WindTre 4G/5G) for highest-trust scenarios.

10.3.2 Mobile Proxy Integration: 4G/5G Carrier IP Pools Mobile carrier IPs provide **superior trust scores** due to: dynamic allocation (residential users share pools), NAT infrastructure (many users behind single IP, reducing per-IP request scrutiny), and legitimate device association (smartphone user agents correlate with network type).

10.3.3 CAPTCHA Solving: Anti-Captcha, 2captcha API Integration

Service	Cost per 1000	Speed	Accuracy	Recommendation
Anti-Captcha	$2-3	5-30s	95%+	Primary for reCAPTCHA v2
2captcha	$1-3	10-60s	90%+	Cost-optimized backup
CapSolver	$2-4	2-10s	95%+	Emerging, API-friendly
Human fallback	Variable	30-120s	99%+	Critical acquisitions only

11. Legal & Ethical Considerations

11.1 Regulatory Landscape

11.1.1 Italian Antitrust Precedent: 2025 CoopCulture Fine (€7M) The **April 2025 AGCM enforcement action** establishes that: **platform operators are liable for inadequate bot protection** (€7M fine to CoopCulture), **third-party automation operators are liable for market manipulation** (€13M collective fines to tour operators), and **individual liability extends to organizational leadership** (named executives in enforcement documents).

11.1.2 EU Digital Services Act: Automated Access Prohibitions The **Digital Services Act (DSA)** effective 2024 creates additional compliance obligations for automated access to online platforms, with **transparency requirements for algorithmic decision-making** and **potential liability for systemic risks** including market manipulation.

11.1.3 Personal Use vs. Commercial Resale: Liability Distinction **Personal automation** for individual/family ticket acquisition—while potentially violating platform terms of service—carries substantially lower regulatory risk than **commercial resale operations**. The 2025 enforcement specifically targeted **scalping with markup resale**, not individual convenience automation.

11.2 Risk Mitigation

11.2.1 Jurisdictional Separation: Non-EU Infrastructure Control Control plane infrastructure in **Switzerland, Iceland, or Singapore** creates regulatory distance from AGCM direct enforcement, though international cooperation agreements may still enable legal action.

11.2.2 Identity Compartmentalization: Per-Ticket Purchase Personas **Operational security**: distinct identities for each acquisition (name variation, email forwarding, payment methods), preventing correlation-based detection and limiting exposure from any single compromise.

11.2.3 Financial OpSec: Cryptocurrency Settlement Where Possible **Payment trail minimization**: cryptocurrency for third-party service payments (proxies, CAPTCHA solving, infrastructure), prepaid cards for ticket purchases where accepted, and corporate structures (LLCs, offshore entities) for operational separation.

12. Full Implementation Reference

12.1 Rust Core: High-Frequency Monitor

Complete implementation available at: `github.com/example/colosseo-monitor-rs` (conceptual reference)

Key modules:

- `src/client.rs`: Fingerprinted HTTP client with JA3 rotation
- `src/parser.rs`: HTML availability extraction with multi-strategy fallback
- `src/state_machine.rs`: Ticket acquisition lifecycle management
- `src/redis.rs`: Distributed coordination and state persistence
- `src/grpc_server.rs`: Service interface for orchestration integration

12.2 Go Orchestrator: Distributed Coordination

Complete implementation available at: `github.com/example/colosseo-orchestrator-go` (conceptual reference)

Key modules:

- `cmd/orchestrator/main.go`: Service entry point with hot-reload
- `internal/proxy/manager.go`: Dynamic proxy pool with health checking
- `internal/notify/dispatcher.go`: Multi-channel notification pipeline
- `internal/config/manager.go`: Viper-based configuration management

12.3 Payment Handler: 3D Secure Automation

Architecture: **hybrid automated/manual** with virtual card API primary, human escalation for challenge flows, and transaction-level idempotency for failure recovery.

12.4 Deployment Manifests: Kubernetes + Docker Compose

Production deployment: **Hetzner/OVH Kubernetes** with auto-scaling monitor pools, Redis-backed state store, and Prometheus/Grafana observability.

Edge deployment: **Raspberry Pi 4 cluster** with Docker Compose, 4G failover, and residential proxy integration for cost-optimized operation.