Architectural Requirements

Quality Requirements Addressed: Link

Architectural Patterns

In developing the BRAD (Bot to Report Abusive Domains) system, several architectural patterns have been chosen to support the project's critical quality requirements.

Gatekeeper Pattern

The Gatekeeper Pattern is implemented in BRAD as a dedicated security layer that mediates all incoming traffic to the system. This component acts as a centralized entry point that handles user authentication, enforces role-based access control (RBAC), and verifies that each request meets the system's security and compliance policies. By introducing this pattern, BRAD directly addresses security, reliability, and compliance requirements. Unauthorized or malformed requests are blocked before reaching sensitive backend services such as the scraper or AI classifier. The Gatekeeper also integrates multi-factor authentication (MFA) for investigator accounts, further securing access to forensic data. In high-traffic or adversarial scenarios, it supports rate limiting, input sanitization, and logging to mitigate threats such as denial-of-service (DoS) attacks or injection attempts.

Quality Requirements Addressed:

1. **Security**: All requests are authenticated, authorized, and validated before reaching internal services, preventing unauthorized access and injection attacks.
2. **Reliability**: The Gatekeeper handles rate limiting, failover routing, and input filtering to protect internal services from overload or failure.
3. **Compliance**: Every access attempt is logged and checked against regulatory rules, ensuring adherence to GDPR and POPIA obligations.

Event-Driven Architecture (EDA)

The Event-Driven Architecture (EDA) enables BRAD to process large volumes of domain investigation requests by allowing system components to operate asynchronously in response to discrete events. When a user submits a suspicious URL, this event triggers a pipeline of subsequent actions such as scraping, malware scanning, AI-based risk scoring, and report generation each executed by specialized services. This pattern enhances scalability by enabling horizontal scaling of event consumers, allowing the system to handle multiple investigations concurrently. It also improves performance by decoupling producers (e.g., the submission module) from consumers (e.g., the scraper bot), enabling real-time, non-blocking execution. Reliability is also addressed through persistent event logs, which ensure that failed or interrupted processes can be recovered and replayed without data loss.

Quality Requirements Addressed:

1. **Scalability**: New event consumers can be added horizontally to meet increased demand during peak investigation periods.
2. **Performance**: Asynchronous processing enables faster throughput for multiple domain investigations running in parallel.
3. **Reliability**: Persistent queues and event logs allow retries and recovery if services fail mid-process.

Service-Oriented Architecture (SOA)

The Service-Oriented Architecture (SOA) is used to decompose BRAD into modular services such as Scrape-Service, Analyse-Service, and Report-Service, each responsible for a well-defined function. These services interact via standardized RESTful APIs, allowing them to operate independently and be developed, deployed, or scaled without disrupting the system as a whole. This directly improves maintainability, as updates or bug fixes in one service do not cascade into others. It also improves interoperability, enabling future integration with external systems such as threat intelligence databases or registrar reporting interfaces. Moreover, scalability is enhanced by the ability to scale only the services under load (e.g., multiple scraper instances during high-volume submissions) rather than the entire system.

Quality Requirements Addressed:

1. **Scalability**: Each service can be scaled based on load without affecting the others.
2. **Maintainability**: Services can be independently updated or replaced, supporting long-term evolution.
3. **Interoperability**: Services follow standard formats and protocols (e.g., JSON, HTTP), enabling integration with external threat intel APIs.

Micro-services Architecture

The Micro-services Architecture builds on SOA by containerizing each service using Docker. Every component whether it's the AI classifier, scraper bot, or report builder is packaged and deployed as a separate micro-service, running in isolated environments. This structure directly improves portability, as each micro-service can be run consistently across local, staging, and production environments. It also enhances fault tolerance and maintainability, because issues in one micro-service (e.g., an AI crash) are isolated from others and can be fixed or redeployed independently. Finally, micro-services improve scalability by enabling fine-grained control over the system’s resource usage, ensuring efficient operation under varying load conditions.

Quality Requirements Addressed:

1. **Scalability**: Each micro-service (e.g., a scraping worker) can be replicated independently for load distribution.

2. **Maintainability**: Faults or updates are isolated to a single service, minimizing system-wide impact.
3. **Portability**: Docker containers ensure that services run reliably across different environments (development, testing, deployment).

Client-Server Model

The Client-Server Model is employed in BRAD to separate the frontend interfaces (client) from backend processing (server). The frontend includes the public user submission portal and the investigator dashboard, both of which communicate with backend services over secured APIs. This pattern strengthens security, as all critical logic and sensitive data processing are centralized on the server, where access can be tightly controlled. It also supports compliance by allowing the server to enforce data validation, consent mechanisms, and logging in accordance with regulations like POPIA and GDPR. Additionally, the model enhances usability, as the client can be optimized for user experience without compromising backend integrity.

Quality Requirements Addressed:

1. **Usability**: A clear separation between UI and backend enables focused UX design for investigators and general users.
2. **Security**: The server centralizes sensitive operations, enforcing access control and API authentication.
3. **Compliance**: Client submissions are sanitized and validated on the server to meet data protection regulations.

Layered Architecture

The system adopts a Layered Architecture to structure its internal logic into four distinct layers: the presentation layer (UI), application layer (API gateway and authentication), business logic layer (scraping and risk analysis), and data layer (databases and logs). This separation improves maintainability, because changes in one layer (e.g., updating the UI) do not ripple across unrelated layers. It also supports security by isolating sensitive logic in backend layers that are not exposed to users. Furthermore, reliability is strengthened, as each layer is testable in isolation, reducing the likelihood of cascading failures.

Quality Requirements Addressed:

1. **Maintainability**: Developers can make changes to one layer (e.g., UI) without affecting others.
2. **Security**: Sensitive operations are encapsulated in deeper layers, reducing attack surface.
3. **Reliability**: Layered isolation makes failures easier to contain and debug.

Pipe and Filter Pattern

The Pipe and Filter Pattern underpins BRAD’s core investigation pipeline, where data flows through a series of processing components (filters), each performing a specific task in the investigation pipeline:

Scrape → Detect Malware → AI Risk Analysis → Metadata Logging → Report Generation

Each component (filter) transforms the input and passes it along the pipeline. This modular design improves maintainability, as filters can be added, replaced, or removed without redesigning the entire flow. It also improves reliability by allowing error handling and fallback mechanisms at each stage. Additionally, performance benefits from clearly defined processing stages, which can be parallelized or scaled independently when needed.

Quality Requirements Addressed:

1. **Maintainability**: Each step can be updated or replaced independently.
2. **Reliability**: The pipeline can resume at failed steps without reprocessing the entire chain.
3. **Performance**: Processing is streamlined through well-defined input/output interfaces.

Model-View-Controller (MVC)

On the frontend, the Model-View-Controller (MVC) pattern is applied to the investigator dashboard to cleanly separate concerns. The model holds domain data and system state, the view renders the UI (e.g., graphs, logs, alerts), and the controller handles user input and orchestrates responses. This structure enhances usability by ensuring that the interface is responsive and intuitive. It also improves maintainability, as frontend developers can update visual components, logic, or data handling independently reducing the likelihood of bugs and simplifying testing.

Quality Requirements Addressed:

1. **Usability**: MVC supports responsive, interactive UIs.
2. **Maintainability**: Clearly separated concerns improve code modularity and testability.

Together, these architectural patterns form a unified blueprint for BRAD’s development. Each pattern was chosen not only for technical elegance, but for its direct and measurable impact on the system’s critical quality requirements. This ensures that BRAD is not only functional but secure, adaptable, and resilient in the face of ever-evolving cyber-security threats.

Design Patterns

1. Factory Pattern

- **Use Case**: Creating different types of bot agents or report objects depending on domain content (e.g., malware, phishing, scam).
- **Benefit**: Encapsulates object creation, improves scalability when new domain types are introduced.

2. Strategy Pattern

- **Use Case**: Switching between scraping techniques (e.g., simple scraper vs. headless browser) or classification models.
- **Benefit**: Makes it easy to plug in new algorithms or scraping methods without altering the core logic.

3. Observer Pattern

- **Use Case**: Real-time alerting system notify investigators when a high-risk domain is flagged.
- **Benefit**: Decouples alert logic from the classification engine.

4. Singleton Pattern

- **Use Case**: Global configuration manager (e.g., for API keys, ML model paths, threat intelligence feeds).
- **Benefit**: Ensures a single point of configuration and avoids conflicting settings.

5. Decorator Pattern

- **Use Case**: Enriching domain reports dynamically with new metadata like threat score, WHOIS, SSL info, etc.
- **Benefit**: Adds functionality without modifying existing report structures.

6. Command Pattern

- **Use Case**: Encapsulating user actions like "submit report," "analyse domain," "override AI decision" as objects.
- **Benefit**: Supports undo, logging, and replay features.

7. Builder Pattern

- **Use Case**: Constructing complex domain reports step by step (text, screenshots, metadata, scores).
- **Benefit**: Separates construction logic from representation.

8. Chain of Responsibility Pattern

- **Use Case**: Processing a domain through a pipeline (e.g., scraping → analysis → risk scoring → report generation).
- **Benefit**: Each step handles the task it's responsible for or passes it to the next step.

9. Adapter Pattern

- **Use Case**: Integrating with various external threat intelligence APIs or WHOIS lookup tools.
- **Benefit**: Converts incompatible interfaces into one that fits your system.

10. Proxy Pattern

- **Use Case**: For secure access to the scraper bot or AI module (e.g., rate-limiting, authentication).
- **Benefit**: Adds a layer of control and security around sensitive components.

11. Mediator Pattern

- **Use Case**: Manages communication between reporters and investigators through an Admin. Reporters submit reports, and the Admin assigns them to available investigators.
- **Benefit**: Prevents direct communication between parties, improves coordination, and keeps the workflow secure and organized.

Architectural Constraints

- **Legal & Compliance Risks**: Must comply with GDPR, POPIA.
- **Domain Blocking & Evasion**: Some sites may block scraping; might require headless browsers or IP rotation. Some websites don’t want to be automatically scanned or scraped by bots. So they use techniques to block your bot from accessing their content. To work around this tools like Headless browsers and IP rotation may be used. They prevent the bot from being blocked by making it seem like it is a normal user when it is fact not a normal user.
- **False Positives in AI Classification**: May require manual override or verification, i.e. AI might incorrectly flag a safe domain as malicious. Since AI isn't perfect, there’s a chance it could make mistakes. That’s why you might need a manual override or human verification, where a security analyst or investigator reviews the case and decides if the AI's decision was actually correct.
- **Data Privacy & Ethics**: Need secure storage, depersonalization, and ethical data handling practices.
- **Budgetary Limits**: Although a server and some funds are provided, the project must stay within the allocated budget.