**System Design Document**

**Backend Developer Assignment**

Chai Shalom Bouchris

**1. Introduction**

The proposed system is a backend platform designed to manage and execute external "black box" algorithms provided by Credito's data team. The system is engineered for scalability, reliability, and flexibility to accommodate the addition of new services or updated versions in the future.

**2. System Architecture**

**2.1. Architecture Diagram**

**A group of white rectangles with black text

Description automatically generated**

**2.2 Component Descriptions**

* **API Gateway:** Acts as the primary entry point for client requests, routing them to the corresponding services while enforcing security and validation measures.
* **Service Orchestrator:** Oversees the execution flow of external services, incorporating error handling, retry mechanisms with exponential backoff, and performance monitoring.
* **Service Registry:** A dynamic repository managing service configurations, including endpoints, parameters, and metadata, enabling flexibility in updating services without system redeployment.
* **Logging & Monitoring Module**: Aggregates logs and performance metrics to ensure system observability, leveraging tools such as Prometheus and Grafana for real-time analytics.
* **Security Module:** Implements comprehensive security controls, including authentication (OAuth2/JWT), authorization, and data encryption.
* **Caching Module (Redis):** Enhances performance by caching frequent API responses and session data, reducing PostgreSQL load, and supporting distributed caching for microservices scalability.

**3. Recommended Technologies**

* **Programming Language**: Java with Spring Boot – Provides a robust platform for developing backend services with built-in capabilities for orchestration, security, and monitoring.
* **API Gateway**: Spring Cloud Gateway – Enables request routing, load balancing, and security features.
* **Service Orchestrator**: Spring Boot with Spring Integration or Spring Cloud Data Flow – Facilitates the management of complex workflows and service orchestration.
* **Service Registry**: Eureka (part of Spring Cloud Netflix) – Supports dynamic service registration and discovery.
* **Logging**: Logback in conjunction with SLF4J – Provides an efficient and flexible logging system.
* **Monitoring**: Prometheus and Grafana – Used for monitoring metrics and displaying dashboards.
* **Security**: Spring Security – Manages authentication, authorization, and data protection.
* **Database**: PostgreSQL – Utilized for storing configurations, logs, and additional data.
* **Caching**: Redis – Used for caching frequently accessed data to improve system performance and reduce database load.
* **Containerization:** Docker with Kubernetes — ensuring scalable, high-availability deployments.

**4. Component Interactions**

**4.1 API Gateway**

* Authentication: Secures incoming requests using OAuth2/JWT.
* Routing: Directs traffic to appropriate microservices.
* Security Hardening: Applies headers to prevent vulnerabilities (e.g., XSS, CSRF).

**4.2 Service Orchestrator**

* Dynamic Loading: Retrieves service configurations from the Service Registry.
* External Calls: Executes REST requests with retry mechanisms. (we can use also gRPC)
* Caching: Utilizes Redis to reduce latency.
* Error Handling: Implements failover strategies for resilience.
* Monitoring: Logs activities and metrics to Prometheus.

**4.3 Service Registry**

* Configuration Management: Manages service endpoints, parameters, and dependencies dynamically.
* Hot Reloading: Supports live updates without downtime.

**4.4 Logging & Monitoring**

* Metrics Collection: Captures request/response cycles, errors, and latency.
* Visualization: Exposes dashboards in Grafana for real-time health tracking.

**4.5 Security Module**

* Authentication & Authorization: Uses Spring Security with OAuth2/JWT tokens.
* Data Protection: Implements encryption protocols for sensitive data.

**4.6 Caching Module (Redis)**

* Performance Optimization: Reduces database load with distributed caching.
* Session Management: Caches session data for faster access.

**5. Design Choices**

**5.1 API Type**

RESTful APIs are employed for intuitive, standardized communication, with potential gRPC integration for high-performance internal service calls.

**5.2 Orchestration**

Spring Integration facilitates seamless orchestration of microservices with support for error handling and complex workflows.

**5.3 Error Handling & Retry Mechanism**

A robust retry mechanism with exponential backoff mitigates service overload risks and increases reliability during transient failures.

**5.4 Monitoring & Logging**

Combining Prometheus with Grafana ensures real-time system health visibility. Logback/SLF4J provides comprehensive log management.

**5.5 Security**

Security is enforced through Spring Security, with OAuth2/JWT for authentication and encrypted data storage.

**6. Scalability and Future Adaptation**

* **Container Orchestration:** Kubernetes supports dynamic scaling based on traffic demands.
* **Service Extensibility:** Adding new services requires minimal code changes—just update the Service Registry.
* **Caching Strategy:** Redis improves throughput by reducing database load.

**7. Integration of Scoring Models**

**While the scoring models' core implementation is out of scope, integration strategies include:**

**7.1 Test-Driven Development (TDD)**

* Pre-defined Test Cases: Covering edge cases and error conditions.
* Unit & Integration Tests: Ensuring system and external service compatibility.
* CI/CD Integration: Continuous automated testing.

**7.2 Using Stubs & Synthetic Models**

* Stubs & Mocks: Simulating external services with predictable responses.
* Synthetic Data: Ensuring privacy-compliant, realistic test data.

**7.3 Production Integration**

* Feature Toggles: Controlled feature rollouts.
* Canary Deployments: Gradual exposure to monitor real-world performance.
* Health Monitoring: Automated checks to detect anomalies.
* Graceful Fallbacks: Ensuring service continuity during failures.

**8. Conclusion**

**The proposed system offers a scalable, secure, and robust architecture designed to efficiently manage external algorithms while supporting future growth. With strong emphasis on observability, security, and extensibility, this design aligns with Credito’s evolving business needs.**

**Evaluation of the Proposed System Design**

**1. Technical Depth**

* **Is the system designed to be scalable and robust?**

Yes, the system is designed with a focus on scalability and robustness. Utilizing Kubernetes for container management allows for the addition and removal of services as needed, enhancing scalability. Additionally, monitoring and logging mechanisms through Prometheus and Grafana assist in real-time issue detection and resolution, contributing to overall system robustness.

* **How does the solution handle errors and manage the lifecycle of each service?**

The system employs orchestration via Spring Integration, enabling the management of complex workflows and flexible error handling. In the event of an external service failure, a retry mechanism with exponential backoff is implemented to prevent service overload and improve the chances of success in subsequent attempts.

**2. Code Quality and Documentation**

* **Is the code clean, modular, and easy to understand?**

The system is designed to be modular, with clear separation between components such as the API Gateway, Service Orchestrator, and Service Registry. This design promotes clean, maintainable code that is easy to understand.

* **Is the documentation clear and helpful?**

The design includes detailed descriptions of each component and an architecture diagram, providing clear and helpful documentation for developers and maintenance teams.

**3. Design and Extensibility**

* **Does the system design allow for adding new services with minimal effort?**

Yes, adding new services is accomplished by updating the Service Registry without requiring changes to the existing codebase, allowing for flexibility and adaptability to evolving needs.

* **Are the chosen technologies and design patterns appropriate for this use case?**

The selected technologies, such as Spring Boot, Spring Integration, and Kubernetes, are well-suited for a system requiring scalability, robustness, and flexibility. The chosen design patterns, including orchestration via Spring Integration, support the system's requirements effectively.

**4. Completeness**

* **Does the solution meet all specified requirements?**

Yes, the proposed solution addresses the specified requirements, including creating an API for external services, error handling, monitoring, and easy extensibility for new services.

* **Are logging, monitoring, and retry mechanisms implemented effectively?**

The system utilizes Logback and SLF4J for logging, Prometheus and Grafana for monitoring, and a retry mechanism with exponential backoff for error handling, ensuring effective implementation of these mechanisms.

In summary, the proposed design offers a scalable, robust, and flexible solution, emphasizing code quality, documentation, and the ability to expand and adapt to future requirements.