ChatGPT

Detailed Technical Specification for Regulens AI

This document expands upon the phased technical specification by providing concrete, implementation-oriented details. It is intended to guide junior developers in building each component of the Regulens AI platform. Where appropriate, example schemas, API definitions and pseudocode are provided. This specification should be read alongside the high-level phased tech spec.

1. Data Models and Schemas

1.1 Regulatory Document Metadata

Regulatory documents are ingested from various sources and stored with the following metadata fields:

```
// Table: regulatory_documents
{
  "id": "UUID",                   // Unique identifier for the document
  "source_uri": "string",         // Original URL or file path
  "title": "string",              // Document title
  "issuer": "string",             // Agency (e.g., CFPB, ECB)
  "publication_date": "date",     // Date the document was published
  "ingestion_date": "timestamp", // When the document was crawled
  "document_type": "enum",        // {RULE, GUIDANCE, PRESS_RELEASE,
ENFORCEMENT}
  "domain": "enum",               // {FAIR_LENDING, PRIVACY,
OPER_RESILIENCE, ...}
  "raw_text": "text",             // Plain-text content extracted
  "summary": "text",              // Generated executive summary
  "processed": "boolean"           // Indicates whether downstream NLP tasks
have been executed
}
```

1.2 Policy Document Index

Policy documents are stored similarly, but with an associated vector embedding and hierarchical structure:

```
// Table: policy_documents
{
  "id": "UUID",
  "file_name": "string",
  "uploaded_by": "UUID",           // User ID of uploader
  "upload_date": "timestamp",
  "document_type": "enum",          // {POLICY, PROCEDURE, FORM, TRAINING}
  "raw_text": "text",
  "vectors": "float[]",            // Embedding vector generated by encoder
  "paragraphs": [
    {
      "paragraph_id": "UUID",
      "text": "string",
      "start_offset": "int",
      "end_offset": "int",
      "vector": "float[]"
    },
    ...
  ]
}
```

1.3 Matching Results and Gap Analysis

Matches between regulatory obligations and policy passages are stored in a separate table:

```
// Table: matches
{
  "id": "UUID",
  "regulatory_document_id": "UUID",
  "policy_document_id": "UUID",
  "obligation_text": "text",      // Specific obligation or requirement
extracted
  "policy_paragraph_id": "UUID",
  "similarity_score": "float",     // Cosine similarity or cross-encoder
score
  "label": "enum",                // {MATCHED, PARTIAL, GAP}
  "created_at": "timestamp",
  "reviewed_by": "UUID",          // User ID who reviewed and labeled
  "comments": "text"
}
```

1.4 Tasks and Action Plans

The action-plan module uses a relational model for tasks with hierarchical relationships:

```
// Table: tasks
{
  "id": "UUID",
  "title": "string",
  "description": "text",
  "regulation_id": "UUID",        // Link back to regulatory document
  "due_date": "date",
  "assignee_id": "UUID",
  "status": "enum",               // {TO_DO, IN_PROGRESS, BLOCKED, COMPLETE}
  "priority": "enum",             // {LOW, MEDIUM, HIGH}
  "parent_task_id": "UUID|null",  // For dependencies
  "created_at": "timestamp",
  "updated_at": "timestamp"
}
```

1.5 Incident Reports

Incident reports must capture key attributes required by DORA:

```
// Table: incidents
{
  "id": "UUID",
  "alert_ids": "UUID[]",          // Alerts aggregated into this incident
  "category": "enum",             // {CYBER, SYSTEM_FAILURE, THIRD_PARTY, ...}
  "severity": "enum",             // {LOW, MEDIUM, HIGH, CRITICAL}
  "description": "text",
  "impact": "text",                // Qualitative and quantitative impact
  "detection_time": "timestamp",
  "report_time": "timestamp",      // When report was generated
  "status": "enum",               // {OPEN, INVESTIGATING, RESOLVED, REPORTED}
  "actions_taken": "text",
  "root_cause": "text",
  "resolution_time": "timestamp|null",
  "regulatory_report": "json"      // Structured object for DORA submission
}
```

2. API Specification

APIs are defined using REST principles. All endpoints are prefixed with `/api/v1`. Authentication tokens must be sent via `Authorization: Bearer <token>` header.

2.1 Authentication

Method	Endpoint	Description
POST	`/auth/login`	Authenticate user with username/password; returns JWT.
POST	`/auth/refresh`	Refresh JWT using a refresh token.

2.2 Regulatory Ingestion

Method	Endpoint	Request Body	Response
POST	`/regulations/scrape`	`{ "source_uri": "string" }`	`{ "job_id": "UUID" }` Enqueues scraping job; returns job identifier.

Method	Endpoint	Request Body	Response
GET	`/regulations/{id}`	–	Returns metadata and content for a regulatory document.
GET	`/regulations?issuer=CFPB&domain=fair_lending`	–	Returns paginated list of regulatory documents filtered by issuer and domain.

2.3 Summarization and Classification

Method	Endpoint	Request	Response
POST	`/summarize`	`{ "document_id": "UUID" }`	`{ "summary": "text", "classification": "enum" }` Generates summary and domain classification.
GET	`/summaries/{id}`	–	Returns stored summary for a document.

2.4 Policy Document Upload and Indexing

Method	Endpoint	Request	Response
POST	`/policies/upload`	`multipart/form-data` with file	`{ "document_id": "UUID" }` Uploads a policy document, extracts text, generates embeddings and stores metadata.
GET	`/policies/{id}`	–	Returns metadata and extracted text for a policy document.

2.5 Semantic Matching and Gap Analysis

Method	Endpoint	Request	Response
POST	`/matching/run`	`{ "regulation_id": "UUID" }`	`{ "match_job_id": "UUID" }` Initiates matching between a regulation and all policy documents.
GET	`/matching/{regulation_id}`	–	Returns list of matches with scores and labels.
POST	`/matching/{match_id}/label`	`{ "label": "enum", "comments": "text" }`	Updates the human-review label and comment for a match.

2.6 Tasks and Action Plans

Method	Endpoint	Request	Response
POST	`/tasks`	`{ "title": "string", "description": "string", "due_date": "date", "assignee_id": "UUID", "parent_task_id": "UUID|null" }`	`{ "id": "UUID" }` Creates a task.
GET	`/tasks/{id}`	–	Returns task details.
PUT	`/tasks/{id}`	`{ "status": "enum", "due_date": "date", "assignee_id": "UUID" }`	Updates task.
GET	`/tasks?assignee_id=<UUID>&status=IN_PROGRESS`	–	Returns filtered list of tasks.

2.7 Incident Management

Method	Endpoint	Request	Response
POST	`/incidents/alerts`	`{ "alerts": [ { "timestamp": "timestamp", "source": "string", "severity": "string", "message": "text" }, ... ] }`	`{ "ingested": true }` Receives batch of alerts from SIEM.
POST	`/incidents/aggregate`	–	Triggers incident aggregation, deduplication and classification.
GET	`/incidents/{id}`	–	Returns incident report with status, impact and actions taken.
PUT	`/incidents/{id}`	`{ "status": "enum", "actions_taken": "text", "resolution_time": "timestamp" }`	Updates incident report.

2.8 Conversational Assistant

The conversational assistant will be exposed via a WebSocket or streaming endpoint to support real-time interactions:

Method	Endpoint	Description
POST	`/assistant/query`	Request: `{ "session_id": "UUID", "question": "string" }`; Response: streamed text with citations.
POST	`/assistant/context/reset`	Clears session context.

3. Sequence of Operations

Below is an example sequence illustrating how a new regulation is processed and mapped to internal policies:

1. **Scraping:** A job is triggered to scrape a CFPB press release. The `/regulations/scrape` endpoint enqueues the task, which the scraper worker retrieves and processes.
2. **Storage:** Extracted text and metadata are stored in the `regulatory_documents` table.
3. **Summarization & Classification:** The `/summarize` endpoint is called with the document ID. The summarization service retrieves the raw text, invokes the LLM to generate a summary and classification, then updates the record.
4. **Matching:** A compliance officer initiates matching via `/matching/run`, specifying the regulation ID. The matching service retrieves the regulation’s obligations, then queries the vector store of policy paragraphs. For each obligation, it computes similarity scores and inserts records into the `matches` table.
5. **Gap Analysis:** The system identifies obligations with no high-scoring matches and flags them as gaps (label `GAP`). These gaps automatically generate tasks via the `/tasks` API, assigning them to the relevant teams.
6. **Review:** Compliance officers review matches in the UI, updating labels and adding comments via `/matching/{match_id}/label`. Feedback is recorded for model improvement.
7. **Action Plan:** Tasks are managed via `/tasks` endpoints, with notifications sent via webhooks to project management tools.

4. Implementation Guidelines

4.1 Technology Stack

- **Languages:** Python for data pipelines and ML services; Node.js or Go for ingestion services requiring concurrency; TypeScript/React for the front-end.
- **Frameworks:** FastAPI (Python) for REST APIs; Celery or RabbitMQ for background tasks; gRPC for high-performance inter-service calls; React with Material UI for front-end.
- **Databases:** PostgreSQL for metadata and transactional data; Object storage (S3/Azure Blob) for raw document storage; Vector database (FAISS/Pinecone) for embeddings.
- **AI Models:** Use Hugging Face transformers (e.g., `bert-base-uncased`, `sentence-transformers/all-MiniLM-L6-v2`) for embeddings; summarization using `facebook/bart-large-cnn` or GPT-4 (via API). Fine-tune models where possible on domain-specific corpora.

- **Orchestration:** Docker for containerization; Kubernetes for orchestration; Helm charts for deployment; Terraform for infrastructure provisioning.

4.2 Coding Standards

- Follow PEP 8 guidelines for Python code and Airbnb style guide for JavaScript/TypeScript.
- Write unit tests with >80 % coverage using PyTest or Jest. Include integration tests for APIs.
- Document public functions and classes using docstrings and JSDoc comments.

4.3 Model Evaluation

- **Summarization:** Evaluate using ROUGE-1, ROUGE-2 and ROUGE-L on a curated set of regulatory documents. Target ROUGE-L $\geq$ 0.4.
- **Classification:** Use accuracy, precision, recall and F1 on a labeled dataset of regulation documents; target F1 $\geq$ 0.9.
- **Matching:** Evaluate using precision and recall at top-k (e.g., top-3 paragraphs). Use manual review from subject matter experts as ground truth.

4.4 Security Practices

- Implement input validation and output encoding to prevent injection attacks.
- Use parameterized queries (SQLAlchemy or ORM) to prevent SQL injection.
- Store secrets (API keys, encryption keys) in a secrets manager (AWS Secrets Manager or Azure Key Vault). Do not hardcode secrets in code.
- Implement rate limiting and logging for all endpoints.
- Follow OWASP Top Ten guidelines and perform periodic vulnerability scans.

5. Deployment and Infrastructure

5.1 Environments

- **Development:** Local Docker environment with mocked external services. Use SQLite or local PostgreSQL for metadata; run MinIO for object storage.
- **Staging:** Hosted on a cloud environment (AWS/Azure/GCP) with separate VPC, using managed PostgreSQL and S3. Set up CI/CD pipeline to deploy automatically on merges to the `staging` branch.
- **Production:** Multi-availability-zone deployment with autoscaling. Use Kubernetes cluster with horizontal pod autoscaler. Enable TLS termination at the load balancer. Configure logging and monitoring.

5.2 CI/CD Pipeline

1. **Code Commit:** Developers submit pull requests for code changes. Automated checks run linting, unit tests, and static code analysis (e.g., SonarQube).
2. **Build:** Container images are built using Docker and tagged with commit hashes.
3. **Test:** Integration tests run in a staging environment. If tests pass, the image is promoted to the staging registry.
4. **Deploy:** Helm charts deploy or update the application on the staging cluster. For production, a manual approval step is required.
5. **Monitor:** After deployment, monitor logs and metrics. Roll back if errors exceed threshold.

6. Development Phases Breakdown

The phased plan described in the high-level spec is repeated here with specific deliverables for developers:

Phase 1 – MVP (Weeks 1–12)

- **Setup:**
- Initialize Git repository and project structure.
- Configure Kubernetes cluster (local Kind or Minikube) for development.
- **Implement Scraper and Ingestion Service:**
- Create `ingestion` microservice in Python using FastAPI and Celery. Add support for HTTP (requests) and PDF parsing (pdfminer.six).
- Use PostgreSQL migration tool (Alembic) to create `regulatory_documents` table.
- **Develop Summarization Service:**
- Implement summarization endpoint that calls external GPT API or runs BART model locally. Cache summaries in PostgreSQL.
- **Build Front-End MVP:**
- Set up React project; implement login page, regulatory list view and detail page.
- **Policy Upload:**
- Build file upload endpoint; parse documents with Tika; generate embeddings; store in FAISS.
- **Basic Matching:**
- Implement matching endpoint using cosine similarity; store results in `matches` table.
- **Deliverable:** Working prototype; demonstration script; development runbook.

Phase 2 – Enhanced Platform (Weeks 13–28)

- **Extend Scraper Sources:** Add support for new websites and RSS feeds; implement parser for multi-language documents.
- **Improved Matching:** Train cross-encoder model using `sentence-transformers/msmarco-MiniLM` and labeled pairs from Phase 1. Update matching endpoint to support cross-encoder scoring.
- **Action-Plan Module:** Create `tasks` table and implement CRUD APIs. Develop UI for task management.
- **Data Virtualization:** Set up Trino; create connectors to sample databases; define unified schema. Provide sample SQL queries and API endpoints to retrieve joined data.
- **Incident Module (Alpha):** Implement ingestion of alerts via POST endpoint; group alerts by hash and timestamp; persist in `alerts` table; provide API to view aggregated alerts.
- **Conversational Assistant (Beta):** Integrate LangChain with GPT; index regulatory summaries and policy paragraphs; implement streaming responses.
- **Security Features:** Add JWT authentication and role check middleware; store hashed passwords using Argon2.
- **Deliverable:** Staging deployment; user guide; integration tests; metrics report.

Phase 3 – Full Productization (Weeks 29–52)

- **Incident Module (Full):** Implement machine-learning classifier for alert severity; support ingestion from Splunk via Kafka; implement DORA reporting template generator.
- **Advanced Data Virtualization:** Integrate additional connectors; implement caching layer; secure virtualization service with certificate-based authentication.

- **Workflow Automation:** Integrate Camunda/Temporal for tasks; support multi-step approval flows.
- **Conversational Assistant (Production):** Deploy model behind an inference API; implement context windows and memory; add support for multiple languages.
- **Multi-Tenancy:** Implement tenant isolation in database and storage; assign tenant IDs to resources.
- **Monitoring and Logging:** Deploy ELK stack and Prometheus/Grafana. Instrument services with metrics and alerts.
- **Compliance Documentation:** Prepare SOC 2 or ISO 27001 artifacts; implement data-processing agreements; compile audit trails.
- **Deliverable:** Production-ready release; full documentation and training materials.

7. Example Pseudocode for Matching

Below is example pseudocode illustrating how the semantic matching engine might operate. This can be translated directly into Python:

```
from sentence_transformers import SentenceTransformer
from sklearn.metrics.pairwise import cosine_similarity

def match_obligations(reg_texts, policy_embeddings, policy_paragraphs,
threshold=0.6):
    """
    reg_texts: List[str]         # list of regulatory obligation sentences
    policy_embeddings: np.array  # matrix of embeddings for paragraphs
    policy_paragraphs: List[str] # list of paragraphs corresponding to
embeddings
    threshold: float             # similarity threshold for a match
    return: List[dict]           # list of match results
    """
    model = SentenceTransformer('all-MiniLM-L6-v2')
    results = []
    for obligation in reg_texts:
        # compute embedding for the obligation
        emb = model.encode([obligation])  # shape: (1, dim)
        # compute cosine similarity with all policy paragraphs
        sims = cosine_similarity(emb, policy_embeddings)[0]
        # find indices where similarity exceeds threshold
        top_idx = sims.argsort()[-5:][::-1]  # top 5 matches
        for idx in top_idx:
            score = sims[idx]
            if score < threshold:
                continue
            results.append({
                'obligation': obligation,
                'paragraph': policy_paragraphs[idx],
                'score': float(score),
            })
    return results
```

This pseudocode demonstrates a simple matching algorithm using cosine similarity. In production, a cross-encoder model may be substituted for improved precision; nonetheless, the above is a starting point for junior developers.

8. Additional Considerations

- **Localization:** When handling multi-language documents, store language codes and use translation services (e.g., `googletrans` or Azure Translator) to normalize documents into a canonical language for embedding.
- **Caching:** Implement caching at multiple layers (HTTP caching for API responses, in-memory caching for repeated queries) to improve performance.
- **Scalability:** For compute-intensive tasks like summarization, consider using serverless inference (e.g., AWS Lambda) or GPU instances behind a queue. Load test each microservice to determine scaling factors.
- **Ethical AI:** Ensure that models are monitored for bias and fairness, especially when processing demographic data required by Section 1071【603437932716430†L165-L239】. Implement a review process where outputs affecting high-impact decisions are validated by human experts.

9. Conclusion

This detailed technical specification breaks down the Regulens AI platform into concrete schemas, API endpoints, sequence diagrams and implementation guidelines. It translates the conceptual design into actionable tasks that junior developers can follow. By adhering to these specifications and iteratively refining through feedback and testing, the development team can implement a robust and scalable compliance platform that meets the requirements identified in the discovery and high-level tech specs.
