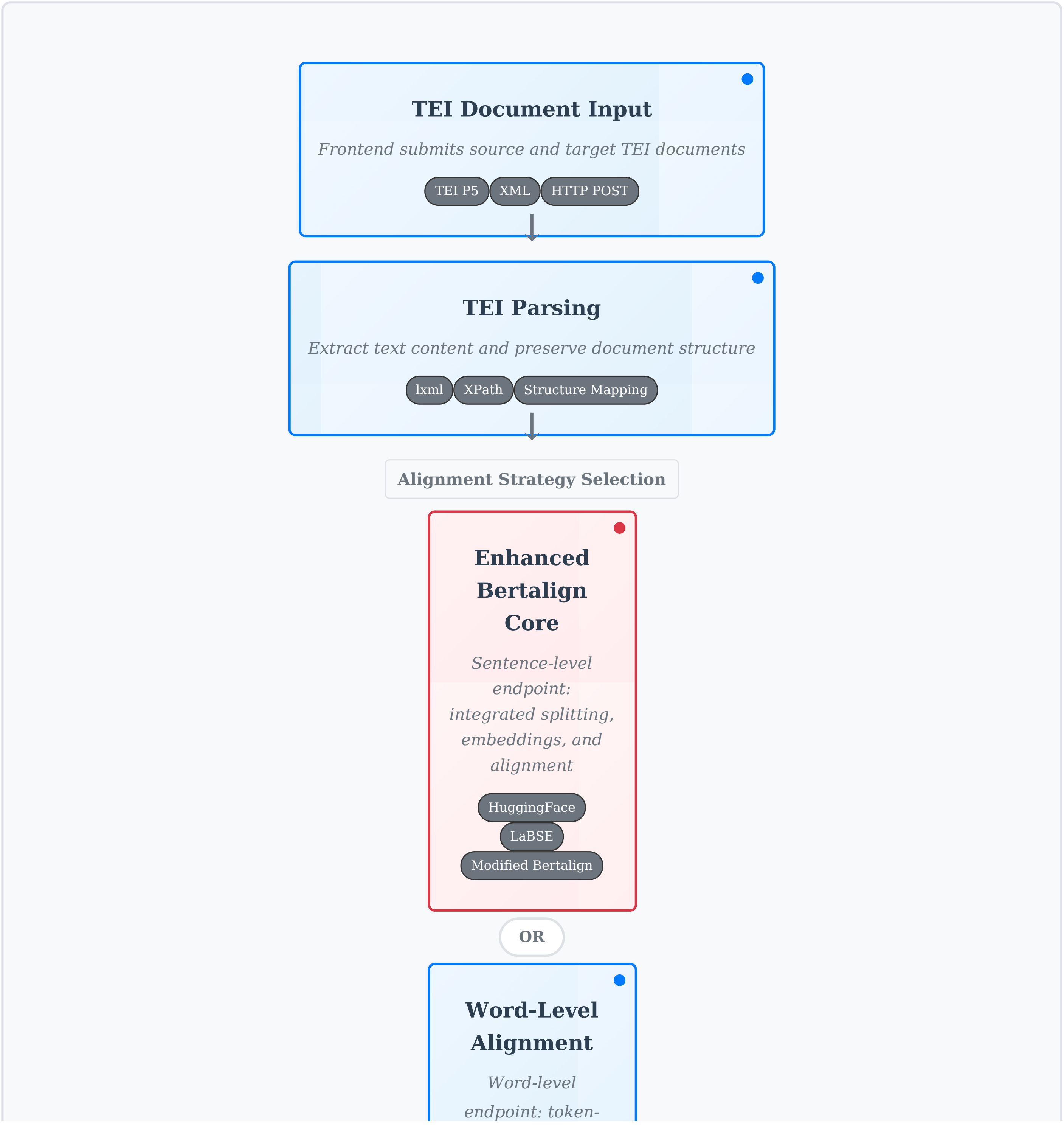
Backend Application for Automatic Translation Alignment

DiScEPT Project - Technical Architecture & Implementation Strategy

Implemented | In Progress | Planned | Optimization Target

Complete Processing Pipeline

End-to-end workflow from TEI input to aligned TEI output with alternative alignment strategies via different API endpoints



TEI Document Input

Frontend submits source and target TEI documents

TEI P5 | XML | HTTP POST

TEI Parsing

Extract text content and preserve document structure

lxml | XPath | Structure Mapping

Alignment Strategy Selection

Enhanced Bertalign Core

Sentence-level endpoint: integrated splitting, embeddings, and alignment

HuggingFace | LaBSE | Modified Bertalign

OR

Word-Level Alignment

Word-level endpoint: token-

based alignment processing

Word Embeddings
Token Alignment

→

TEI Reconstruction

Generate aligned TEI with linkGrp annotations

lxml.builder linkGrp TEI P5

↓

Response Delivery

Return aligned TEI document to frontend

FastAPI JSON Response

System Architecture Layers

API Gateway Layer

Implemented

FastAPI Application

Asynchronous web framework with automatic OpenAPI documentation, request validation, and error handling

FastAPI Uvicorn Pydantic OpenAPI

Request Validation

Input validation and sanitization using Pydantic models for alignment request payloads

Pydantic Models Type Validation

Rate Limiting & Authentication

Request throttling and API key management for production deployment

Redis slowapi JWT

TEI Processing Layer

Next Implementation Priority

TEI Document Parser

Comprehensive TEI P5 XML parsing with metadata extraction, structure preservation, and validation

lxml | TEI P5 Schema | XPath | XML Validation

Text Extraction & Mapping

Extract plain text content while maintaining precise position mapping to original TEI structure

Text Extraction | Position Mapping | Structure Preservation

TEI Reconstructor

Rebuild TEI documents with alignment annotations using linkGrp elements according to TEI P5 standards

lxml.builder | linkGrp Generation | TEI Compliance

Alignment Processing Layer

Under Optimization

Enhanced Bertalign Core

Rebuilt Bertalign with integrated HuggingFace embeddings generation, removing external dependencies

Modified Bertalign | HuggingFace API | LaBSE Embeddings | Internal Sentence Splitting

Optimization Strategy

Removing googletrans dependency and integrating HuggingFace API calls directly into alignment logic

Word-Level Alignment Module

Separate processing component for word-level alignment, operating independently from Bertalign

Word Embeddings | Token Alignment | Similarity Metrics

Caching System

Redis-based caching for embeddings and alignment results to improve performance

Redis | Embedding Cache | TTL Management

Infrastructure Layer

Google Cloud Deployed

Google Cloud Run Deployment

Containerized deployment with auto-scaling, utilizing Google Cloud Build for optimized Docker images

Docker | Cloud Run | Cloud Build | Auto-scaling

CI/CD Pipeline

Automated deployment pipeline using GitHub Actions to build and deploy to Google Cloud Run on code changes

GitHub Actions | CI/CD | Automated Deployment | Git Integration

Model Deployment Strategy

Lazy loading of models at deployment stage with HuggingFace API integration for reduced image size

Lazy Loading | HuggingFace API | Container Optimization

Frontend Integration Requirements

Critical components needed for complete system integration between frontend and backend services

Frontend Request Handler

Integration of alignment service calls into existing frontend infrastructure

Progress Tracking Interface

Real-time progress indicators for long-running alignment processes, including status updates and estimated completion times

Error Handling & Feedback

Comprehensive error messaging and user feedback system for failed alignment requests or invalid TEI document submissions

Technical Requirements

System specifications and deployment considerations for the DiScEPT backend application

Current Deployment: Google Cloud Run

Containerized deployment with automatic scaling, handling Docker images up to 10GB with lazy model loading at startup

Docker Container | Auto-scaling | Cloud Build | Lazy Loading

Minimum Server Requirements

For potential Institute server deployment: 8GB RAM minimum, GPU recommended for local model inference, Docker support required

8GB+ RAM | GPU Optional | Docker Engine | Python 3.9+

Network & Storage Requirements

HuggingFace API access required, Redis for caching, persistent storage for model cache and temporary processing files

Internet Access | Redis Server | Persistent Storage | HTTPS Support

Critical Architecture Questions

Immediate Prototype Deployment

Should we continue with Google Cloud Run for prototype development and testing, or migrate to Institute servers now?

Long-term Distribution Strategy

What's our vision for service distribution after prototype validation: (1) Open source release where users self-deploy, (2) Hosted service on Institute infrastructure, (3) CLARIN infrastructure deployment for European academic access, or (4) Multiple deployment options? This determines our development priorities and sustainability model.

Document Length Constraints

What are the practical length limits for TEI documents submitted from the frontend?

TEI Format Coverage & Scope

Should we focus exclusively on the current TEI P5 format we're using, or do we need to support multiple TEI schema variations and encoding models? What's the scope of TEI diversity we expect from our users?

Language Selection Strategy

Should we implement a predefined language selection dropdown in the frontend where users manually choose source and target languages? This would eliminate the need for automatic language detection (googletrans dependency) and significantly reduce Docker image size.

Impact: *Docker image optimization, dependency reduction, user workflow simplicity, potential user errors from incorrect language selection*

User Base & Concurrent Usage Expectations

For normal prototype usage, we expect 1-2 concurrent users, but workshops and demonstrations could bring sudden spikes of 10-20+ simultaneous requests. Should we implement a queue-based processing system (similar to eScriptorium) to handle these peak loads gracefully, or rely on auto-scaling with potential wait times during demos?

Impact: *User experience during demos, infrastructure costs during idle periods, system complexity, workshop planning requirements*

Data Storage & Privacy

What are the privacy requirements for handling potentially unpublished academic texts or sensitive historical documents? Should we store textual data before and after alignment to evaluate algorithm performance, or must all processing be ephemeral?

Bertalign Parameter Configuration

Should we expose Bertalign's configuration parameters (max_align, top_k, win, skip, margin, len_penalty) to users through the frontend interface, or define optimal default values internally? User control offers flexibility for different text types and languages, but increases interface complexity and requires parameter expertise.

Impact: *User interface complexity, alignment quality control, parameter expertise requirements, system predictability*

Frontend Segmentation Implementation

With frontend-based segmentation decided, what are the implementation requirements? Which JavaScript tokenization libraries work best cross-linguistically? Should we use web workers for performance? How do we ensure consistent word/sentence boundaries across different languages and text types while maintaining TEI structure preservation?

Impact: *Frontend performance, linguistic accuracy, client-side resource requirements, cross-language compatibility*

Manual-Automatic Alignment Integration

How should the system handle the transition between automatic and manual alignment modes? Should users be able to trigger automatic alignment after beginning manual corrections, and how do we merge or override existing manual alignments? What happens to user corrections when automatic alignment is re-run?

Impact: *User workflow design, data preservation strategy, alignment quality assurance, undo/redo functionality requirements*

TEI Format & Identifier Strategy

Should the backend generate TEI documents that exactly match the frontend's manual alignment format (UUID identifiers, standOff structure, word-level markup)? Is the current TEI alignment schema considered final, or should we design for schema evolution? Are UUIDs the optimal identifier strategy considering document size vs. uniqueness trade-offs?

Impact: *Frontend-backend consistency, long-term compatibility, document file size, migration complexity*