**SYSTEM LOGO**

# Ancient Egyptian Language Management System (AELMS)

**System Design Document**
**Version 1.0**
Feb. 01, 2025

# 1. Introduction

## 1.1 Purpose:

Goal of the document and what it aims to achieve

This document describes and tracks the necessary information required to effectively define the architecture and system design for the AELMS to guide developers, stakeholders, and other project participants on how the system will be developed and maintained. This design document will be incrementally and iteratively produced during the system development life cycle, based on the particular circumstances of the system and the system development methodology used.

## 1.2 Scope:

Boundaries of the software system being designed, including what is and isn't included in the scope

### 1.2.1 In Scope:

The system will be used by two categories of users (customers and operators) through different frontends to reach different backend microservices generally divided into customer, operator, and shared services.

The system will generally allow customer users to:

1. View **public or landing pages** encouraging them to log in or register for a time-bound **free trial**.
2. View, interact with, opine on or critique, listen to, cross-reference, and inspect **resources** (glyphs, words, classics) and their **linguistic elements** (definitions, pronunciation, phonetics, sample-sentences, appendices, word-glossaries, and classic-lines).
3. Maintain a collection of **bookmarked words and classic-lines**.
4. Receive **in-app notifications** on newly added and edited resources.
5. Use **location-based services** such as Google Maps within the app to view the location of classics on a live map without providing directions or redirecting to the Google Maps application. *A static map with pointer calculated based on classic location coordinates may be used as placeholders in case of Google Maps API unavailability or as a means to reduce production cost and external API dependency.*
6. Use **language and UI localization** options, and other accessibility and usability approaches to maximize UX.
7. Manage a **personal user account** and make **payment** using payment processing service such as Stripe Payment.

8. Receive **email correspondence** on registration email verification, subscription status, and MFA through ESPs such as SendGrid.
9. Receive support through **chatbot interaction**, **support page**, and **issue logs**.

The system will generally allow operator users to:

1. **Create and manage resources** (glyphs, words, definitions, classics, and classic-lines). And use Google Translate in a **multilingual translation** for resources. *The use of Google Translate API may be provided only as an option to resource operators to limit production cost and dependence on external APIs.*
2. Issue **in-app resource notifications**.
3. **Audit resources** performance and edits, user accounts, and issue-logs to **generate business reports**.
4. Provide **personalized support to users** that is tracked using an issue-log and provide email correspondence on issue-log using the SendGrid ESP.
5. **Manage all accounts** (customers and/or operators) and billings
6. View and **manage a personal operator profile**.

**

### 1.2.2 Out of Scope:

Requirements identified during analysis but not included in the current project scope include:

- **audio narration and video illustrations** for classics,
- **games and quizzes**, and
- **user groups**.

## 1.4 References:

Provides links or references to related documentation, such as system requirements or other relevant materials

- Stripe API Documentation
- SendGrid v3 API Documentation
- Google Maps Documentation
- Google Translate Documentation
- Google Typescript Style Guide
- Google Javascript Style Guide
- TS Dev Style Guide
- Twelve Factor App
- Twelve-Factor App on ECS and Fargate
- OSWAP Security Cheatsheet
- OSWAP NodeJS Security Cheat Sheet
- NodeJS Threat Model
- Kuiwan Javascript Security Guide
- Typescript Documentation
- Amazon ECS Overview
- Amazon ECS Developer Guide
- Amazon ECS section of AWS CLI Reference
- Amazon ECS API Reference

## 1.5 Overview:

A high-level summary of the software system, highlighting key components and how they interact

The AELMS will be built mostly using the **MERN technology tool stack** and mostly **JS programming language** (and TS a superset of JS) for both **frontend and backend services**. The system will be structured as a distributed system using a **microservices architecture** with the presentation and services (an associated databases) components individually managed using **containerization**. The microservices will use the **circuit-breaker pattern** to decouple services and increase resilience to individual service failure. To reduces operational overheads, databases used by each service will be manage externally using **cloud-based database** services rather than been containerized and self-deployed.

The containerized services will be managed by a **container orchestration engine** in conjunction with an **ALB** that scales the services independently in response to traffic. All public traffic to the application through the ALB will be received and managed using an **API Gateway** in conjunction with an **IdP** to provide system-wide user authentication and authorization. A **DNS** will be used to manage public addresses to the services and a **CDN** will be used together with both frontends to manage static asset caching. Few shared **in-memory caches** will be deployed along-side the services to cache

responses from expensive database queries. There is no presently identified need for the use of an **event broker** or **event queue** to manage asynchronous inter-service communication.

The app will use **logging and monitoring tools** with the orchestration engine to store and analyze **container activities**. The chat service may be implemented using **Python programming language** to leverage its machine learning libraries and third-party APIs and a different backend framework centered around Python.

The system will use two frontends:

(1) **Customer-facing**: Interfaces between customers (and visitors) and **customer backend services**,

(2) **Operator-facing**: Interfaces between operators and **operator services**.

The customer frontend will be presented through a web browsers on both desktop and mobile **and later on iOS and Android devices**. The operator frontend will only be presented through web browsers and optimized for viewing only on desktop devices.

The **Customer frontend** will serve some **server-side rendered webpages** using **SSR** and static-site frameworks like **NextJS** (An extension of **ReactJS**) to cater to **SEO** and fast initial page load for visiting users.

The system will use a **CI/CD pipeline** to maintain **development-production parity** to keep development, staging, and production as similar as possible. For local development, the system will use Docker for containerization and Docker Compose for orchestration. For production, the microservices will be deployed on AWS cloud ecosystem using **AWS ECS** for orchestration and **AWS Fargate** for serverless container compute. The system's implementation will closely adhere to the **twelve-factor app methodologies** for **SaaS applications** working together as a distributed system and other recommended cloud-based development principles.