**Программная архитектура системы редактирования и валидации метрологических стандартов**

**Обзор системы**

Разрабатываемая система представляет собой комплексное решение для автоматизированной обработки, редактирования и валидации метрологических стандартов и ГОСТов. Система построена на основе микросервисной архитектуры и предназначена для использования экспертами в области метрологии, обеспечивая стандартизированный подход к оформлению нормативных документов с перспективой интеграции искусственного интеллекта для семантического анализа содержимого.

**Архитектурные решения и их обоснование**

**Выбор микросервисной архитектуры**

Принятие решения о использовании микросервисной архитектуры обусловлено несколькими ключевыми факторами, характерными для данной предметной области:

Разделение ответственности по функциональным доменам. Система четко разграничивает три основных области: управление пользователями и профилями, хранение и управление документами, а также специализированная обработка документов. Такое разделение позволяет каждому сервису фокусироваться на своей core-компетенции, что особенно важно для сложной задачи форматирования документов по стандартам ГОСТ.

Технологическая гетерогенность. Различные компоненты системы имеют принципиально разные требования к производительности и функциональности. Сервис обработки документов требует богатых библиотек для работы с форматами Office, что делает Python оптимальным выбором благодаря таким инструментам как python-docx. В то же время сервисы профилей и хранения документов ориентированы на высокую производительность и конкурентность, что обосновывает использование Go с его эффективной моделью goroutines и встроенной поддержкой HTTP/2.

Масштабируемость и изоляция отказов. В контексте обработки больших объемов документов критически важна возможность независимого масштабирования компонентов. Операции форматирования документов являются ресурсоемкими и могут требовать значительного времени выполнения, в то время как операции с профилями пользователей должны выполняться с минимальной задержкой.

**Технологический стек и его обоснование**

Backend-сервисы на Go выбраны для компонентов, требующих высокой производительности при работе с concurrent запросами. Go обеспечивает эффективную работу с базами данных благодаря развитой экосистеме ORM (GORM) и нативной поддержке пула соединений. Статическая типизация и компиляция в нативный код критически важны для production-ready решений в enterprise-среде.

Python для сервиса обработки документов обусловлен наличием специализированных библиотек для работы с форматом DOCX и возможностью интеграции с ML-моделями для будущего семантического анализа. Фреймворк Flask обеспечивает быстрое прототипирование и развертывание REST API с минимальными накладными расходами.

PostgreSQL как основное хранилище выбран за счет превосходной поддержки ACID-транзакций, расширенных индексов (включая полнотекстовый поиск для метаданных документов) и возможности хранения JSON-структур для гибких пользовательских профилей. Репликация и партиционирование PostgreSQL критически важны для обеспечения высокой доступности системы.

MinIO для объектного хранения представляет собой S3-совместимое решение, позволяющее эффективно управлять версионированием документов, обеспечивать распределение данных и интегрироваться с CDN для быстрой доставки контента.

**Межсервисное взаимодействие и паттерны надежности**

REST API как протокол взаимодействия. Выбор REST API обусловлен его универсальностью и простотой отладки, что критически важно на этапе активной разработки. HTTP/JSON обеспечивает прозрачность взаимодействия и упрощает мониторинг системы.

Однако такой подход требует применения специальных паттернов для обеспечения надежности:

Circuit Breaker Pattern реализуется для предотвращения каскадных отказов при недоступности сервиса обработки документов. Если сервис форматирования перегружен или недоступен, circuit breaker переводит систему в режим деградации, предотвращая накопление неотвеченных запросов.

Retry с экспоненциальным backoff применяется для обработки временных сбоев сети, особенно критичных при работе с объектным хранилищем. Стратегия повторных попыток учитывает специфику операций форматирования, которые могут занимать продолжительное время.

Timeout Configuration настраивается дифференцированно: короткие таймауты для операций с профилями (< 1с), средние для загрузки документов (< 10с), и расширенные для операций форматирования (< 60с).

**Обработка состояний и идемпотентность**

Idempotent Operations Design особенно важен для операций форматирования документов. Каждый запрос на обработку получает уникальный идентификатор, позволяющий безопасно повторять операции без дублирования результата.

**Контейнеризация и развертывание**

Использование Docker и Docker Compose обеспечивает консистентность сред разработки и production, что критически важно для системы с гетерогенным технологическим стеком. Каждый микросервис упаковывается в отдельный контейнер с минимальным базовым образом (Alpine Linux), что снижает поверхность атак и ускоряет развертывание.

Multi-stage builds применяются для оптимизации размера образов, особенно для Go-сервисов, где финальный образ содержит только скомпилированный бинарный файл без инструментов сборки.

**Вопросы масштабируемости и производительности**

Текущая архитектура закладывает основу для горизонтального масштабирования, однако требует дальнейшего развития:

Stateless Service Design обеспечивает возможность запуска множественных экземпляров каждого сервиса за load balancer. Особое внимание уделено сервису обработки документов, где состояние операции сохраняется во внешнем хранилище.

Database Connection Pooling реализован для оптимизации работы с PostgreSQL, с учетом специфики Go и Python приложений. Connection pool настраивается исходя из ожидаемой нагрузки и характеристик операций.

Планируемое добавление API Gateway позволит реализовать централизованную аутентификацию, rate limiting и мониторинг, что является следующим логичным шагом в развитии архитектуры.

**Интеграция с искусственным интеллектом для комплексной валидации документов**

Архитектура системы изначально проектируется с учетом интеграции специализированных AI-модулей для автоматизированной валидации метрологических документов по четырем ключевым направлениям. Выбор Python в качестве основы для сервиса обработки документов обусловлен не только наличием библиотек для работы с DOCX, но и богатой экосистемой машинного обучения, включающей PyTorch, Transformers, и spaCy.

Система будет детектировать дублирование ГОСТов через семантическое сравнение на основе векторных моделей, обученных на метрологических текстах. Каждый раздел документа преобразуется в вектор и сравнится с базой стандартов с использованием FAISS для быстрого поиска. Иерархическая проверка выявит полные совпадения, копирование разделов или отдельных формулировок, а временной анализ с применением графовых нейросетей учтёт версионность документов, отличая обновления от плагиата.

Правовая валидация будет проводиться моделью, обученной на законах и регламентах, которая проверит соответствие документа действующему законодательству через извлечение ссылок (NER) и оценку соответствия (compliance scoring). Терминологическая корректность обеспечится графовой моделью и дообученным GPT для контекстного анализа, а проверка смысловой целостности выявит логические несоответствия. Система будет автоматизировать экспертизу документов с возможностью ручного контроля и постоянного дообучения.