ENGINEERING REFERENCE: Abstract Wiki Architect (v1.0)

System Type: Distributed Natural Language Generation (NLG) Platform

Architecture Pattern: Hexagonal (Ports & Adapters) + Event-Driven

Core Stack: Python 3.12+, FastAPI, Redis, Docker, Grammatical Framework (GF)

---

1. Architectural Philosophy

The system explicitly rejects the "Big Ball of Mud" anti-pattern common in academic linguistic software. Instead, it enforces strict **Domain-Driven Design (DDD)** boundaries to ensure the linguistic core remains isolated from infrastructure volatility.

1.1 Hexagonal Isolation

- **The Domain (Inner Hexagon):** Located in `app/core`. Contains pure business logic (`Language`, `Frame`, `Sentence`) and `SystemEvent` definitions. It has **zero dependencies** on external libraries (framework-agnostic).
- **The Ports:** Defined in `app/core/ports`. Abstract Base Classes (ABCs) that define the contracts for persistence (`LanguageRepo`), messaging (`EventBroker`), and generation (`GrammarEngine`).
- **The Adapters (Outer Hexagon):** Located in `app/adapters`.
    - *Driving Adapters:* `FastAPI` (REST Interface), `CLI`.
    - *Driven Adapters:* `RedisBroker` (Messaging), `WikidataAdapter` (External Data), `GfGrammarEngine` (NLG Implementation), `FileSystemRepo` (Persistence).

Benefit: We can swap the storage from FileSystem to PostgreSQL, or the Messaging from Redis to Kafka, without touching a single line of the linguistic generation logic.

2. The NLG Pipeline (The "Engine")

The core differentiator is the integration of **Symbolic AI** (Grammatical Framework) rather than purely Statistical AI (LLMs). This ensures 100% grammatical correctness and hallucination-free output.

2.1 The Abstract Syntax Tree (AST) Flow

Data flows through the system in four stages:

1. **Ingestion (JSON Frame):** The API receives a Semantic Frame (e.g., `{"pred": "Love", "subj": "John", "obj": "Mary"}`).
2. **Enrichment (Wikidata Sagas):**
    - The system queries Wikidata SPARQL endpoints to resolve Q-IDs (e.g., `Q42` -> `Douglas Adams`).
    - *Resilience:* This adapter is wrapped in **Circuit Breakers** (via `tenacity`) to handle upstream Wikidata outages gracefully.
3. **Abstract Syntax Mapping:**
    - The enriched frame is mapped to a GF Abstract Syntax Tree.
    - *Example:* `PredVP (UsePN John) (ComplSlash (SlashV2a Love_V2) (UsePN Mary))`
4. **Linearization (PGF Runtime):**
    - The system calls the **PGF (Portable Grammatical Framework)** binary.
    - It applies the specific **Concrete Grammar** (e.g., `src_finnish.txt`) to render the surface string.
    - *Morphology Handling:* The engine automatically handles case inflection (e.g., turning "Mary" into the accusative case depending on the verb "Love").

3. Concurrency & Scalability Model

The system is designed for high-throughput, non-blocking operations, critical when compiling massive grammar files.

3.1 Asynchronous Core

- **FastAPI & Uvicorn:** The HTTP layer runs on an `asyncio` loop, handling thousands of concurrent idle connections.
- **Non-blocking I/O:** All external calls (Wikidata, Redis) use async drivers (`httpx`, `redis-py`).

3.2 The "Worker" Model

Grammar compilation (converting `.gf` source to `.pgf` binary) is a CPU-intensive operation that can take 10-60 seconds per language.

- **Decoupling:** The API *never* compiles grammars. It pushes a `COMPILE_REQUEST` job to Redis.
- **Task Queue:** Implemented using **`arq`** (a lightweight, async wrapper around Redis Streams).
- **Scaling Strategy:** The `Worker` container (`docker/Dockerfile.worker`) is stateless. You can horizontally scale the number of worker replicas to increase compilation throughput linearly.

3.3 Event Sourcing (Lite)

The system publishes domain events (`LANGUAGE_ONBOARDED`, `LEXICON_UPDATED`) to a Redis Pub/Sub channel. This allows for future extensions (e.g., a Notification Service or Analytics Service) to subscribe to events without tight coupling.

4. Linguistic Engineering Implementation

This is the high-value IP. The system does not use simple string replacement templates; it uses full computational grammars.

4.1 Concrete Grammar Complexity

The uploaded assets (src/) demonstrate "Industrial-Grade" implementation of high-complexity languages:

- **Finnish (src_finnish):** Implements full agglutinative morphology (15 noun cases).
- **Arabic (src_arabic):** Implements Semitic root-and-pattern morphology (interdigitating roots).
- **Basque (src_basque):** Handles ergative-absolutive alignment (where the subject of an intransitive verb behaves like the object of a transitive verb).

4.2 The "Morphodict" Bridge

Located in src/morphodict. This acts as the **Lexical Interface**.

- It maps Abstract Concepts (Wikidata Q-Items) to the specific inflectional paradigms (Lemmas) in the RGL (Resource Grammar Library).
- *Optimization:* The build pipeline allows for "Hot Swapping" of lexicons without recompiling the entire grammatical structure.

5. Quality Assurance Infrastructure

The system employs a "Trust but Verify" approach to linguistic generation.

5.1 Universal Dependencies (UD) Validation

- **The Problem:** How do you know the generated Finnish is correct if you don't speak Finnish?
- **The Solution:** The CI pipeline (`.github/workflows`) executes `treebanks.txt` scenarios.
- **Mechanism:**
    - Generate text via GF.
    - Parse the text using a standard dependency parser (e.g., UDPipe).
    - Compare the resulting CoNLL-U trees against the expected "Gold Standard."
    - *Verdict:* This provides mathematical proof of grammatical correctness.

5.2 Integration Testing

- **Harness:** `gf_test.py` and `run_tests.py`.
- **Scope:** Runs regression tests on the `GfGrammarEngine` adapter to ensure that changes in the abstract syntax do not break concrete linearizations in any of the 6+ supported languages.

6. Deployment & Operations

6.1 Containerization

- **Multi-Stage Builds:** `Dockerfile.backend` and `Dockerfile.worker` use multi-stage builds to minimize image size, discarding build-time dependencies (like Haskell compilers) from the runtime image where possible.
- **Volumes:** The `app/data` directory is mounted as a volume, persisting the `FileSystemRepo` state across container restarts.

6.2 Configuration

- **12-Factor App:** All configuration is injected via Environment Variables (`.env` -> `app/shared/config.py`).
- **Pydantic Settings:** Configuration is strictly typed. The app will fail to start if `REDIS_URL` is missing or malformed, preventing runtime errors.

7. Current Technical Limitations (To be addressed)

1. **Cold Start Latency:** The PGF runtime must load the .pgf file into memory. For massive grammars (morphodict + RGL), this can consume significant RAM (500MB+ per worker).
2. **Lexical Coverage:** While the *grammar* is complete, the *lexicon* depends on the morphodict. If a Q-Item is missing, the system falls back to the Q-ID string (e.g., "Q42 lives in Paris").
3. **Authentication:** The API currently lacks an Authorization Bearer layer (OAuth2/JWT). This is a trivial addition for a production rollout (app/adapters/api/dependencies.py).