Apodeixi Technical Direction

# Current State

Apodeixi 1.0 was completed in December 2021.

It provides adequate functionality for early adopters but suffers from some shortcomings. In particular:

* Slow performance
* Over-complex coding required when adding new manifest kinds. There is too much boilerplate code and care is needed to know which kind of boilerplate code is needed when.
* Potentially brittle under core changes (e.g., to the parser). In spite of a good suite of tests, it requires a lot of understanding of all of Apodeixi to correctly evolve a part of it.

Much of this arises from Apodeixi 1.0 being a monolith as well as from the explorative nature of its development: functionality was only identified along the development of Apodeixi 1.0, so some coding patterns are overly verbose and inefficient/complex, hence brittle.

The parser is a good example of this: it was conceived as parsing one manifest at a time, when in reality there is a need for cross-manifest relationships within a posting (e.g., joins). This led to complex callbacks at certain points in the flow that go back to concrete kind-specific controllers.

More generally, controllers are full of special code to assist generic engines with some manipulation specific to a manifest kind. Some of this is unnecessary, if the engines are re-ideated in a more generic manner. Moreover, the “special code” is not adhering to only one pattern: sometimes it is “forward” pattern and sometimes it’s a “callback” pattern:

* “Forward” pattern refers to concrete controllers creating certain config objects that are passed as parameters to the engine. It is a way for controllers to summarize some settings in a “generic” schema so that the engine’s behavior can be altered based on settings without the engine needing to know what manifest kind it is. We call this “forward” because the role of the controller is at the start of the engine flow.
* “Callback” pattern refers to the fact that at certain points in the core engine’s flow, the engine needs some logic that is specific to a manifest kind, and for that it delegates to a callback in the controller.

As a result of this, the code of controllers is not easy to understand to the uninitiated, since depending on the code (i.e., controller method), it will be called at different flows in the engine, either “early” (before entering the “generic engine” processing) or in the middle of such “generic engine” processing (as for callbacks).

# Strategic Direction

The path forward is to make two evolutions of Apodeixi:

* Apodeixi 2.0: re-write Apodeixi using a services architecture
* Apodeixi 3.0: migrate Apodeixi 2.0’ services to a K8s deployment model.

Apodeixi 3.0 is the target state.

However, we don’t pursue K8s immediately to avoid getting distracted with learning the minutia of how to handle K8s deployments.

Instead, Apodeixi 2.0 is an intermediate state where the monolith is decomposed into “services” which are “independently deployable”. While these independent deployments are not yet based on containerization or K8s, they are very analogous to containers and is expected that the Apodeixi 2.0 architecture can be naturally migrated to K8s.

The rest of this document discusses Apodeixi 2.0’s architectural approach.

# Apodeixi 2.0 Services

## Service Domain Model

Apodeixi 2.0 decomposes Apodeixi 1.0’s capabilities per bounded context, and each bounded context results in a service.

It is envisioned that Apodeixi 2.0 will consist of the following services:

* **Posting service**: processes an Excel posting to produce a set of YAML artifacts.
* **Form service**: creates Excel spreadsheets that can be used to support postings.
* **Report service**: creates Excel reports and Pandas DataFrames based on the data held in Apodeixi.
* **Import services**: for each source system, a service to import data.
* **Store service**: persistence “backing service” to save YAML manifests
* **KnowledgeBase service**: a gateway that receives traffic from the outside world, and re-routes requests to one of the other services as appropriate.
* **Operator service**: an internal technical administration service, used (for example) to manage the lifecycle and inter-service communication of service instances. In Apodeixi 3.0 this will be replaced by K8s mechanisms, but for Apodeixi 2.0 this handcrafted, minimalist service is needed in the absence of K8s.

While not technically a “service”, these two other deployable components complete the picture:

* **CLI**: a fat client command-line interface that sends requests to the KnowledgeBase service
* **SDK**: a Python module that has functions used to invoke various Apodeixi services and retrieve data suitable for a programmer. For example, methods to get joined manifests as a Pandas DataFrame object, for subsequent manipulation by the programmer. If imported in a Python execution environment (e.g., in Jupyter Notebook), then a developer can manipulate data extracted from Apodeixi and create further analysis or export it to downstream systems.

## Pythonic Coding Standard

Apodeixi 2.0 breaks the Apodeixi 1.0 monolith into separate services.

Additionally, Apodeixi 2.0 embraces a Pythonic standard. This means that much of the boilerplate in Apodeixi 1.0 will be eliminated by more succinct Python idioms, leveraging things like:

* Iterators and generators
* Context managers
* Decorators

In particular, boilerplate arising from cross-cutting concerns needs to be eliminated by an appropriate Pythonic idiom, to enforce separation of concerns in the codebase.

## Microservice Approach

Each Apodeixi 2.0 service is a microservice, in that:

* Each service is independently deployable
* Each service runs as a separate operation-system process
* Each service can only be invoked though a REST API

Apodeixi 2.0 also adopts some simplifications on the microservice construct:

* All services and all clients (e.g., CLI and SDK) run in the same machine, i.e., localhost
* There is only 1 user making 1 synchronous request at a time (so no concurrency)
* There is only 1 instance of each type of service.

Thus, while Apodeixi 2.0 is a distributed system architecture, from the end-user perspective it looks like Apodeixi 1.0 in the sense of looking like a desktop application.

It is expected that these simplifications make it easier to implement the Apodeixi 2.0 Operator service, so that design focus for Apodeixi 2.0 is on breaking up the monolith into services, as opposed to the complexities of distributed systems.

The vision is that distributed system complexities can be allowed in Apodeixi 3.0, which would leverage K8s technologies to take care of such complexities.

## Repo per Service

Each Apodeixi 2.0 service will have a dedicated source code repo, with a unique Python project in such repo.

This is a deliberate tactic in order to break the Apodeixi 1.0 monolith: if multiple services had their code in the same source code repo and same Python project, experience shows that developers invariantly end up coupling both services. Typically, this arises as a convenience during development: while implementing some functionality in service A, a developer may find it convenient to re-use some code in service B by simply making a function call to a Python module in B.

To put some rails against this “easy but damaging coupling”, we make it impossible for a service’s codebase to invoke code from another service. That is why each service has its own repo.

## Conda environment per Service

Each Apodeixi 2.0 service runs in a dedicated Conda environment.

This ensures that each service owns its 3rd party dependencies, i.e., the list and versions of 3rd party Python modules it depends on.

This is healthy because:

* It makes it easier to track 3rd party dependencies, since each service’s Conda environment can have the minimalist set of dependencies needed by that one service
* Makes it possible for services to have mutually inconsistent dependencies (i.e., avoids the equivalent of “JAR hell”
* Makes it possible to upgrade dependencies on a per-service basis.

## Deployment “image” for a service

Apodeixi 2.0 does not use containers.

However, it uses something “similar” to containers as the unit of deployment granularity.

For each service, its “deployment” runtime consists of:

* A Python executable running on Windows in localhost
* Running a REST server whose code comes entirely from that service’s repo
* With all 3rd parties coming from that service’s Conda environment.

## Performance & a Data Cache

In Apodeixi 1.0, each user request leads to the launch of a new Python process for all of Apodeixi to run in it.

As a result, each request needs to again retrieve data from the store and put it in memory. It is suspected this marshalling of data is partly responsible for the slow performance of Apodeixi 1.0.

With Apodeixi 2.0’s service architecture, service instances survive across user requests. In fact, service instances only terminate when explicitly shut down by the Operator service or if there is fatal failure.

Thus, it becomes possible in Apodeixi 2.0 to deploy a data cache, so that data accessed in one request can be more quickly available in future requests.

It is expected that this should speed up performance relative to Apodeixi 1.0, even though there will be “hops” in traversing services in Apodeixi 2.0. The performance penalty of “hops” will be kept minimal by having a “thin” API for services, i.e., exchange as little data as possible and minimize costs of marshalling data whenever possible.

# Blueprints and Proof Points

Before development can start, some blueprints need to be completed to validate the design approach that must be followed.

## Iterator-based Pythonic Parser

A prototype of the parser must be implemented to show how a Pythonic approach would look.

## REST services: API, discovery and deployment

A prototype of a few services must be implemented to show who services can be stood up and invoked.

## Test Harness

A prototype is needed to verify that the test harness of Apodeixi 1.0 can be leveraged in testing Apodeixi 2.0.

Potentially, changes to the test harness will be required. For example, to distinguish:

* “unit” tests, which test an algorithm (e.g., a Python module)
* “service” tests, which test a service and its API, probably using stubs for service dependencies
* “end-to-end” tests, which test from the user experience (e.g., the CLI)