Folio Architecture: How It Works Under the Hood

This document explains how Folio works internally. It's written for users with basic Python knowledge who want to understand the codebase well enough to modify or extend it.

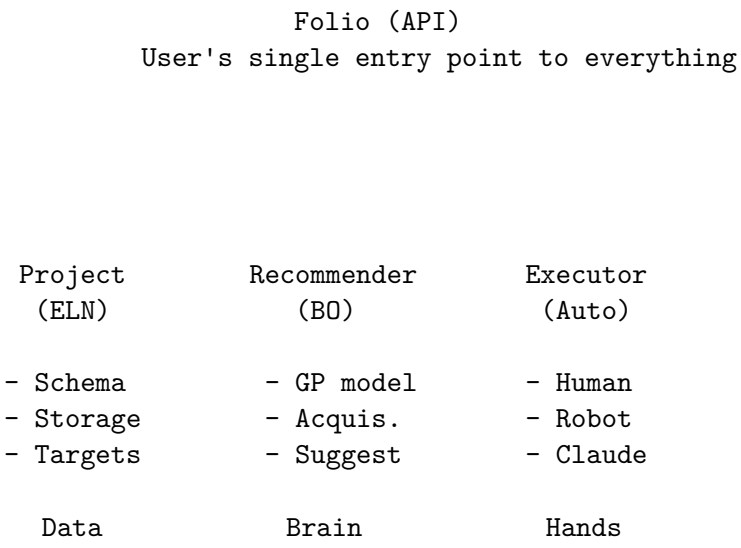
Table of Contents

The Big Picture

Folio follows a simple loop:

```
Record experiment results → Model learns from data → Suggest next experiment → Repeat
```

This is implemented through three main components:



Project: Stores what your experiment looks like (inputs, outputs) and all your data (observations).

Recommender: The "brain" that looks at your data and suggests what to try next.

Executor: Optionally runs experiments automatically (via human prompts, robot API, or Claude-Light).

Directory Structure

```
src/folio/
   api.py              # Folio class - the main entry point users interact with
   exceptions.py       # Custom error types (ProjectNotFoundError, etc.)

   core/               # Data structures and storage
      schema.py        # InputSpec, OutputSpec - define what inputs/outputs look like
      observation.py   # Observation - one experiment result
      project.py       # Project - experiment schema + settings
      config.py        # TargetConfig, RecommenderConfig - settings objects
      database.py      # Database - SQLite storage and retrieval

   recommenders/       # Suggestion generation
      base.py          # Recommender ABC - interface all recommenders follow
      bayesian.py      # BayesianRecommender - GP + acquisition optimization
      random.py        # RandomRecommender - random sampling baseline
      acquisitions/    # Acquisition functions (EI, UCB, NEHVI)

   surrogates/         # Models that learn from data
      base.py          # Surrogate ABC - interface all surrogates follow
      gp.py            # SingleTaskGPSurrogate - Gaussian Process
      multitask_gp.py  # MultiTaskGPSurrogate - for multi-objective

   targets/            # How to compute optimization targets from outputs
      base.py          # ScalarTarget ABC
      builtin.py       # DirectTarget, RatioTarget, etc.

   executors/          # Automated experiment running
       base.py         # Executor ABC
       human.py        # HumanExecutor - CLI prompts
       claude_light.py # ClaudeLightExecutor - autonomous API
```

Core Data Classes

InputSpec and OutputSpec (schema.py)

These define what your experiment's inputs and outputs look like:

```python
@dataclass
class InputSpec:
    name: str                           # "temperature"
    type: str                           # "continuous" or "categorical"
    bounds: tuple[float, float] | None  # (20.0, 100.0) for continuous
    levels: list[str] | None            # ["low", "high"] for categorical
    units: str | None                   # "°C"
    optimizable: bool = True            # False = context variable

@dataclass
class OutputSpec:
    name: str           # "yield"
    units: str | None   # "%"
```

Observation (observation.py)

One experiment result:

```python
@dataclass
class Observation:
    project_id: int                     # Which project this belongs to
    inputs: dict[str, float | str]      # {"temperature": 80, "solvent": "THF"}
    outputs: dict[str, float]           # {"yield": 72.5}
    timestamp: datetime                 # When recorded (auto-set)
    notes: str | None                   # Free-form notes
    tag: str | None                     # Grouping label
    failed: bool = False                # Mark failed experiments
    id: int | None = None               # Database ID (auto-set)
```

Project (project.py)

The experiment schema plus configuration:

```
@dataclass
class Project:
    id: int
    name: str
    inputs: list[InputSpec]            # What inputs the experiment takes
    outputs: list[OutputSpec]          # What outputs it produces
    target_configs: list[TargetConfig] # What to optimize (can be multiple)
    reference_point: list[float] | None # For multi-objective (Pareto)
    recommender_config: RecommenderConfig  # Which recommender to use
```

TargetConfig (config.py)

Defines what to optimize:

```
@dataclass
class TargetConfig:
    objective: str              # "yield" - which output to optimize
    objective_mode: str         # "maximize" or "minimize"
    target_type: str            # "direct", "ratio", "difference", etc.
    # Additional fields for derived targets (numerator, denominator, etc.)
```

RecommenderConfig (config.py)

Settings for the recommendation algorithm:

```
@dataclass
class RecommenderConfig:
    type: str = "bayesian"      # "bayesian" or "random"
    surrogate: str = "gp"       # "gp" or "multitask_gp"
    acquisition: str = "ei"     # "ei", "ucb", "variance"
    mo_acquisition: str = "nehvi"  # For multi-objective
    n_initial: int = 3          # Random samples before using GP
    acquisition_kwargs: dict    # Extra params like {"beta": 2.0}
```

The Main Loop: From Observation to Suggestion

Here's what happens in a typical Folio session:

1. Create a Folio Instance

```
folio = Folio("experiments.db")
```

What happens: - Creates/opens SQLite database at `experiments.db` - `Folio` object holds reference to `Database` object - `Folio._recommenders = {}` cache is initialized (empty)

2. Create a Project

```
folio.create_project(
    name="my_experiment",
    inputs=[InputSpec("x", "continuous", bounds=(0, 10))],
    outputs=[OutputSpec("y")],
    target_configs=[TargetConfig("y", objective_mode="maximize")],
)
```

What happens:

```
Folio.create_project()

       Validate inputs (bounds exist for continuous, etc.)
       Create Project object
       Database.create_project(project)
           INSERT INTO projects (name, inputs_json, outputs_json, ...)
```

3. Add Observations

```
folio.add_observation("my_experiment",
    inputs={"x": 5.0},
    outputs={"y": 12.3})
```

What happens:

```
Folio.add_observation()

       Database.get_project("my_experiment")
           SELECT * FROM projects WHERE name = ?

       Validate inputs match project schema
       Create Observation object (timestamp auto-set)

       Database.add_observation(observation)
           INSERT INTO observations (project_id, inputs_json, outputs_json, ...)
```

4. Get Suggestions

```
suggestions = folio.suggest("my_experiment")
```

What happens:

```
Folio.suggest()

       Database.get_project("my_experiment")
       Database.get_observations(project_id)

       Get or create recommender:
        if "my_experiment" not in self._recommenders:
            self._build_recommender(project)
                   Creates BayesianRecommender(project)

       recommender.recommend(observations)

           project.get_training_data(observations)

              For each observation:
                - Extract input values → row in X
                - Compute targets → row in y

              Returns X shape (n, d), y shape (n, m)

           project.get_optimization_bounds()
              Returns bounds shape (2, d)
```

```
recommender.recommend_from_data(X, y, bounds, maximize)

    (Inside BayesianRecommender)

    If len(X) < n_initial:
         return random_sample_from_bounds(bounds)

    _fit_surrogate(X, y)
        Train GP model on data

    _build_acquisition(X, y, maximize)
        Create EI/UCB acquisition function

    _optimize_acquisition(bounds)
        Find X that maximizes acquisition
            Returns suggested input values
```

Class-by-Class Breakdown

Folio (api.py)

The main class users interact with. It's a thin wrapper that coordinates everything:

```python
class Folio:
    def __init__(self, db_path: str):
        self._db = Database(db_path)
        self._recommenders: dict[str, Recommender] = {}

    # Project CRUD
    def create_project(self, name, inputs, outputs, target_configs, ...): ...
    def get_project(self, name) -> Project: ...
    def list_projects(self) -> list[str]: ...
    def delete_project(self, name): ...

    # Observation CRUD
    def add_observation(self, project_name, inputs, outputs, ...): ...
    def get_observations(self, project_name, tag=None) -> list[Observation]: ...
    def delete_observation(self, project_name, observation_id): ...
```

```
    # Recommendation
    def suggest(self, project_name, n=1, fixed_inputs=None) -> list[dict]: ...
    def get_recommender(self, project_name) -> Recommender: ...

    # Execution
    def build_executor(self, executor_type): ...
    def execute(self, project_name, n_iter, executor=None): ...
```

Key point: Folio caches recommenders in `_recommenders` dict. Once created, a recommender is reused for subsequent `suggest()` calls on the same project.

Database (core/database.py)

Handles all SQLite operations:

```
class Database:
    def __init__(self, db_path: str, sync_url=None, auth_token=None):
        # Connect to SQLite (or libSQL for cloud sync)
        self._conn = sqlite3.connect(db_path)
        self._create_tables()

    def _create_tables(self):
        # CREATE TABLE projects (id, name, inputs_json, outputs_json, ...)
        # CREATE TABLE observations (id, project_id, inputs_json, ...)

    # Project operations
    def create_project(self, project: Project) -> int: ...
    def get_project(self, name: str) -> Project: ...
    def get_project_by_id(self, project_id: int) -> Project: ...
    def delete_project(self, name: str): ...

    # Observation operations
    def add_observation(self, obs: Observation) -> int: ...
    def get_observations(self, project_id: int) -> list[Observation]: ...
    def delete_observation(self, obs_id: int): ...
```

Data is stored as JSON: Inputs, outputs, and configs are serialized to JSON strings in the database, then deserialized when loaded.

Project (core/project.py)

Beyond storing schema, Project has methods to extract training data:

```python
class Project:
    def get_training_data(self, observations: list[Observation]) -> tuple[np.ndarray, np.nda
        """Convert observations to X, y arrays for ML.

        Returns
        -------
        X : np.ndarray, shape (n_observations, n_continuous_inputs)
        y : np.ndarray, shape (n_observations, n_targets)
        """
        X = []
        y = []

        for obs in observations:
            if obs.failed:
                continue  # Skip failed experiments

            # Extract continuous input values in order
            row_x = [obs.inputs[inp.name] for inp in self.inputs
                     if inp.type == "continuous"]
            X.append(row_x)

            # Compute target values
            row_y = []
            for target_config in self.target_configs:
                target = self._build_target(target_config)
                row_y.append(target.compute(obs))
            y.append(row_y)

        return np.array(X), np.array(y)

    def get_optimization_bounds(self) -> np.ndarray:
        """Get bounds for optimizable continuous inputs.

        Returns shape (2, n_optimizable): row 0 = lower, row 1 = upper.
        """
        bounds = []
        for inp in self.inputs:
            if inp.type == "continuous" and inp.optimizable:
```

```
                bounds.append(inp.bounds)
        return np.array(bounds).T  # Transpose to (2, d)
```

Recommender (recommenders/base.py)

Abstract base class defining the recommender interface:

```
class Recommender(ABC):
    def __init__(self, project: Project):
        self.project = project

    def recommend(self, observations: list[Observation], fixed_inputs=None) -> dict:
        """High-level: observations in, suggestion dict out."""

        # 1. Extract arrays from observations
        X, y = self.project.get_training_data(observations)
        bounds = self.project.get_optimization_bounds()
        maximize = [tc.objective_mode == "maximize"
                    for tc in self.project.target_configs]

        # 2. Call subclass implementation
        candidate = self.recommend_from_data(X, y, bounds, maximize, ...)

        # 3. Convert array back to dict
        result = {}
        for i, inp in enumerate(self.project.get_optimizable_inputs()):
            result[inp.name] = float(candidate[i])
        return result

    @abstractmethod
    def recommend_from_data(self, X, y, bounds, maximize, ...) -> np.ndarray:
        """Low-level: arrays in, array out. Subclasses implement this."""
        ...

    @staticmethod
    def random_sample_from_bounds(bounds: np.ndarray) -> np.ndarray:
        """Uniform random sample within bounds."""
        lower = bounds[0, :]
        upper = bounds[1, :]
        return np.random.uniform(lower, upper)
```

BayesianRecommender (recommenders/bayesian.py)

The main recommender using Gaussian Process + acquisition function:

```python
class BayesianRecommender(Recommender):
    def __init__(self, project: Project):
        super().__init__(project)
        self._surrogate = None  # GP model, created when needed

    def recommend_from_data(self, X, y, bounds, maximize, ...):
        # Not enough data? Return random sample
        n_initial = self.project.recommender_config.n_initial
        if len(X) < n_initial:
            return self.random_sample_from_bounds(bounds)

        # Fit GP model
        self._fit_surrogate(X, y)

        # Optimize acquisition function
        return self._optimize_acquisition(X, y, bounds, maximize, ...)

    def _fit_surrogate(self, X, y):
        """Create and fit GP model."""
        self._surrogate = self._build_surrogate()
        self._surrogate.fit(X, y)

    def _build_surrogate(self):
        """Create surrogate based on config."""
        surrogate_type = self.project.recommender_config.surrogate
        if surrogate_type == "gp":
            return SingleTaskGPSurrogate()
        elif surrogate_type == "multitask_gp":
            return MultiTaskGPSurrogate(n_tasks=len(self.project.target_configs))
        else:
            raise ValueError(f"Unknown surrogate: {surrogate_type}")

    def _build_acquisition(self, X, y, maximize):
        """Create acquisition function based on config."""
        acq_type = self.project.recommender_config.acquisition
        model = self._surrogate.model  # BoTorch model

        if acq_type == "ei":
            best_f = y.max() if maximize[0] else y.min()
```

```
            return ExpectedImprovement(model, best_f, maximize[0])
        elif acq_type == "ucb":
            beta = self.project.recommender_config.acquisition_kwargs.get("beta", 2.0)
            return UpperConfidenceBound(model, beta, maximize[0])
        # ... more acquisition types

    def _optimize_acquisition(self, X, y, bounds, maximize, ...):
        """Find input that maximizes acquisition function."""
        acq = self._build_acquisition(X, y, maximize)

        # Use BoTorch's optimizer
        candidates, _ = optimize_acqf(
            acq_function=acq,
            bounds=torch.tensor(bounds),
            q=1,
            num_restarts=5,
            raw_samples=20,
        )

        return candidates.numpy().flatten()
```

Surrogate (surrogates/base.py and gp.py)

Surrogates are models that learn from data and make predictions:

```
# base.py
class Surrogate(ABC):
    @abstractmethod
    def fit(self, X: np.ndarray, y: np.ndarray) -> None:
        """Train model on data."""
        ...

    @abstractmethod
    def predict(self, X: np.ndarray) -> tuple[np.ndarray, np.ndarray]:
        """Predict mean and std at X."""
        ...

# gp.py
class SingleTaskGPSurrogate(Surrogate):
    def __init__(self):
        self._model = None
```

```
        self._is_fitted = False

    def fit(self, X, y):
        # Convert to torch tensors
        train_X = torch.tensor(X, dtype=torch.float64)
        train_y = torch.tensor(y, dtype=torch.float64)

        # Create BoTorch GP model
        self._model = SingleTaskGP(train_X, train_y)

        # Fit hyperparameters
        mll = ExactMarginalLogLikelihood(self._model.likelihood, self._model)
        fit_gpytorch_mll(mll)

        self._is_fitted = True

    def predict(self, X):
        test_X = torch.tensor(X, dtype=torch.float64)
        posterior = self._model.posterior(test_X)
        mean = posterior.mean.detach().numpy()
        std = posterior.variance.sqrt().detach().numpy()
        return mean, std

    @property
    def model(self):
        """Access underlying BoTorch model (for acquisition functions)."""
        return self._model
```

Target (targets/base.py and builtin.py)

Targets compute optimization objectives from observation outputs:

```
# base.py
class ScalarTarget(ABC):
    @abstractmethod
    def compute(self, observation: Observation) -> float:
        """Extract scalar target value from observation."""
        ...

# builtin.py
class DirectTarget(ScalarTarget):
```

```
    """Use an output directly as the target."""

    def __init__(self, output_name: str):
        self.output_name = output_name

    def compute(self, observation: Observation) -> float:
        return observation.outputs[self.output_name]

class RatioTarget(ScalarTarget):
    """Target = numerator / denominator."""

    def __init__(self, numerator: str, denominator: str):
        self.numerator = numerator
        self.denominator = denominator

    def compute(self, observation: Observation) -> float:
        num = observation.outputs[self.numerator]
        den = observation.outputs[self.denominator]
        return num / den
```

How Suggestions Are Generated

Here's the full flow when `folio.suggest()` is called with enough data:

```
folio.suggest("my_project")

   Get project and observations from database

   Get cached recommender (or create new BayesianRecommender)

   recommender.recommend(observations)

       project.get_training_data(observations)

           observations = [
               Observation(inputs={"x": 1.0}, outputs={"y": 2.5}),
               Observation(inputs={"x": 5.0}, outputs={"y": 8.2}),
               Observation(inputs={"x": 9.0}, outputs={"y": 5.1}),
           ]
```

```
        X = [[1.0], [5.0], [9.0]]     shape (3, 1)
         y = [[2.5], [8.2], [5.1]]     shape (3, 1)

    bounds = [[0.0], [10.0]]          shape (2, 1)

    recommender.recommend_from_data(X, y, bounds, [True])

        len(X) = 3 >= n_initial = 3, so use GP

        _fit_surrogate(X, y)

            Create SingleTaskGPSurrogate
            surrogate.fit(X, y)

                Create BoTorch SingleTaskGP(train_X, train_y)
                Optimize GP hyperparameters (lengthscale, noise)

            GP now knows: "y peaks around x=5"

        _build_acquisition(X, y, [True])

            best_f = 8.2 (max observed y)
            Create ExpectedImprovement(model, best_f=8.2, maximize=True)

        _optimize_acquisition(bounds)

             Call BoTorch optimize_acqf:
              - Evaluate acquisition at many random points
              - Use L-BFGS-B to find maximum

             Returns [4.8]  (x value with highest expected improvement)

    Convert to dict: {"x": 4.8}

Return [{"x": 4.8}]
```

Extending Folio

Adding a Custom Recommender

1. Create a new file in `recommenders/`:

```python
# recommenders/my_recommender.py
from folio.recommenders.base import Recommender

class MyRecommender(Recommender):
    def recommend_from_data(self, X, y, bounds, maximize,
                            fixed_feature_indices=None,
                            fixed_feature_values=None):
        # Your logic here
        # Must return np.ndarray of shape (n_optimizable_features,)
        ...
```

2. Register it in `BayesianRecommender` or create a factory.

Adding a Custom Surrogate

1. Inherit from `Surrogate`:

```python
# surrogates/my_surrogate.py
from folio.surrogates.base import Surrogate

class MySurrogate(Surrogate):
    def fit(self, X, y):
        # Train your model
        ...

    def predict(self, X):
        # Return (mean, std) arrays
        ...

    @property
    def model(self):
        # Return underlying model for acquisition functions
        ...
```

2. Add to `BayesianRecommender._build_surrogate()`.

Adding a Custom Acquisition Function

1. Create in `recommenders/acquisitions/`:

```python
# recommenders/acquisitions/my_acquisition.py
from botorch.acquisition import AcquisitionFunction

class MyAcquisition(AcquisitionFunction):
    def forward(self, X):
        # Return acquisition values for X
        # Higher = more promising to sample
        ...
```

2. Add to `BayesianRecommender._build_acquisition()`.

Adding a Custom Target

1. Inherit from `ScalarTarget`:

```python
# targets/my_target.py
from folio.targets.base import ScalarTarget

class MyTarget(ScalarTarget):
    def __init__(self, param1, param2):
        self.param1 = param1
        self.param2 = param2

    def compute(self, observation):
        # Calculate scalar from observation.outputs
        return some_calculation(observation.outputs, self.param1, self.param2)
```

2. Register in `Project._build_target()` or use directly.

Adding a Custom Executor

1. Inherit from `Executor`:

```python
# executors/my_executor.py
from folio.executors.base import Executor
from folio.core.observation import Observation

class MyExecutor(Executor):
    def _run(self, suggestion: dict, project) -> Observation:
        # Run experiment (call API, robot, simulation, etc.)
        outputs = my_experiment_function(**suggestion)

        return Observation(
            project_id=project.id,
            inputs=suggestion,
            outputs=outputs,
        )
```

2. Pass to `folio.execute()`:

```python
folio.execute("my_project", n_iter=10, executor=MyExecutor())
```

Summary: Where to Find Things

I want to...	Look in...
Change how data is stored	`core/database.py`
Add new input/output types	`core/schema.py`
Change project validation	`core/project.py`
Add a new recommender	`recommenders/`
Change how GPs work	`surrogates/`
Add acquisition functions	`recommenders/acquisitions/`
Add derived targets	`targets/`
Add automation options	`executors/`
Change the user API	`api.py`