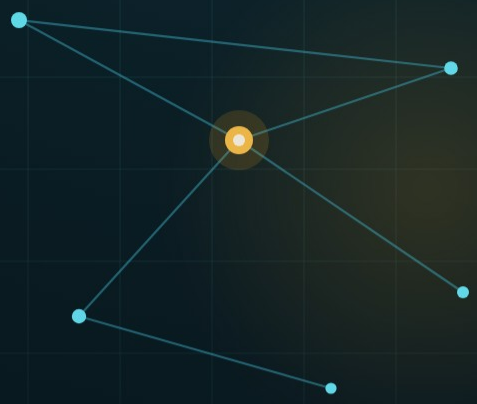
firefly
framework

The Complete Guide · 2026



STRATEGY · ARCHITECTURE · PRACTICE

Firefly DataScience

Turn **AI ambition into governed, production-grade outcomes** — fusing generative AI with proven machine learning, without the risk, the lock-in, or the wait.

Faster time-to-value | Governed GenAI | No vendor lock-in | Production-ready

DOCUMENT

The Complete Guide — Strategy, Architecture & Practice

The Complete Guide — 2026
Firefly Software Foundation · Apache-2.0

Contents

Strategy · Architecture · Practice — for leaders and engineers

PART I

Strategy & the business case

Why AI initiatives stall, the operating model that changes the odds, and the governed, measurable outcomes a leader can expect.

Part I · Strategy · Chapter 1

Executive summary

Firefly DataScience turns AI ambition into governed, production-grade outcomes — fusing generative AI with proven machine learning, without the risk, the lock-in, or the wait.

`fireflyframework-datascience` is an open-source (Apache-2.0) Python metaframework for AutoML. It pairs **generative AI** — built on the Firefly Agentic substrate, which wraps Pydantic AI — with **classical machine learning and Deep Learning**, so any team can apply data science to any project quickly, with production governance, hexagonal swappability, and security by default.

> ◆ **The pattern at its core**
>
> The LLM proposes; a deterministic classical engine decides. GenAI proposes code, features, pipelines and seeds; a classical engine trains, scores and selects; and **every GenAI step is gated behind a measured improvement over a seeded classical baseline**. GenAI is a governed, measurably-gated accelerator over a battle-tested classical core — never a black box.

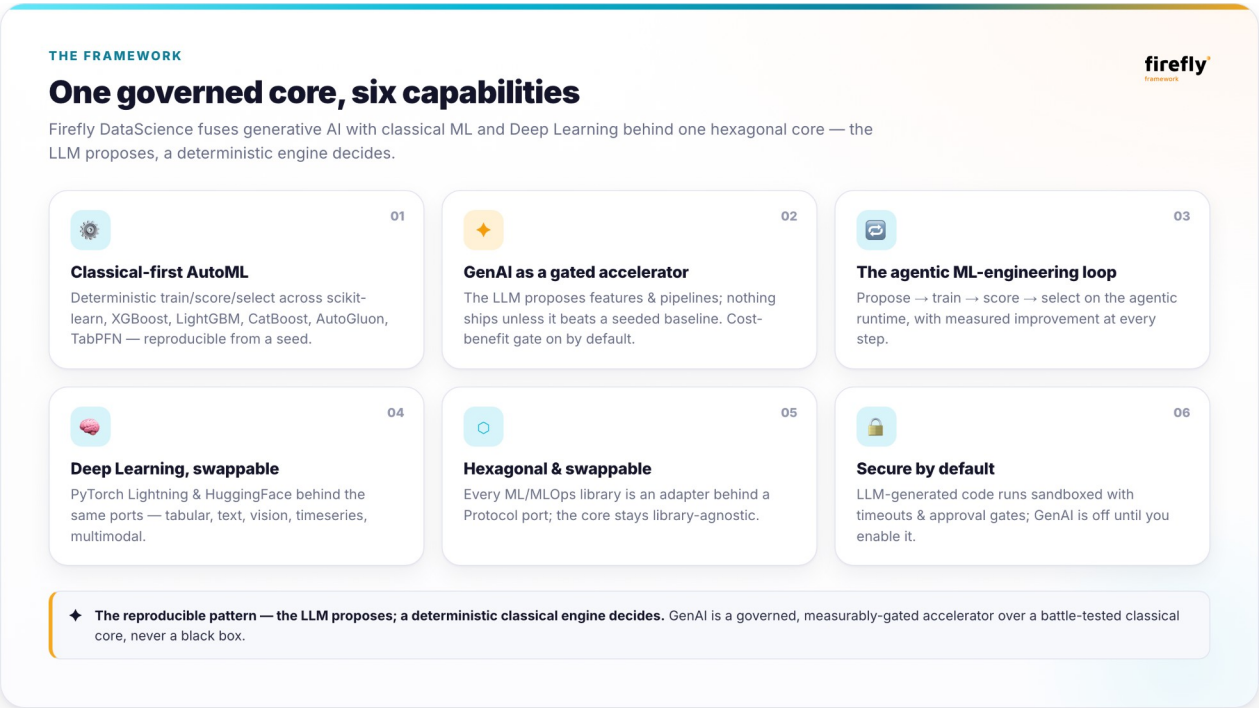


Figure 1. *One governed core, six capabilities — the framework at a glance.*

What it changes

Stripped of the engineering detail, the framework changes five things a leader cares about: it compresses time-to-value from quarters to days, makes generative AI *governed* and auditable, removes vendor and library lock-in, lowers cost and risk, and treats production — serving, lineage, validation, benchmarking — as a first-class concern rather than an afterthought.

What the evidence shows

The claims are measured, not asserted. In an unbiased **nested cross-validation** across five datasets, automated selection beats a single logistic-regression baseline by **+0.029 ROC-AUC** (Wilcoxon $p = 0.046$) and a single XGBoost by **+0.030** ($p \approx 7.5e\text{-}6$), while matching a tuned random forest. In a controlled ablation with a real LLM, GenAI feature engineering lifts a linear model by **+0.0205 ROC-AUC** ($p = 0.0039$) for **under \$0.01** — and the cost-benefit gate guarantees it never regresses the baseline. Out of the box on the public OpenML `credit-g` benchmark, the pipeline reaches **0.82 ROC-AUC**, comparable to leading AutoML tools.

> **✓ Reproducible by construction**
>
> Every number above is produced by a bundled benchmark harness with a fixed seed and default trainers. Runs are seed-pinned, decisions are logged, and lineage is captured end-to-end — so results can be reproduced and audited, not just believed.

Who this guide is for

This document is layered. **Part I** (this part) is for business and transformation leaders: the opportunity, the operating model, the outcomes, and how to start. **Parts II–IV** are for engineers and architects: the hexagonal architecture, the classical and GenAI engines, a complete hands-on tutorial, an end-to-end worked example, and the reference material. Each audience can stop where the detail stops being useful to them.

Part I · Strategy · Chapter 2

Why AI initiatives stall — and the Firefly answer

Most organizations are not short of AI ambition. They are short of AI *in production*.

Boards approve the budgets and teams run the experiments. Yet industry analyses repeatedly find that the majority of machine-learning models never make the journey from a data scientist's notebook to a running system that actually informs a decision. The reasons are rarely about the algorithms. The friction shows up in four familiar places:

- **Bespoke, one-off code.** Each project is hand-built and hard to maintain, so little of the effort compounds.
- **A prototype-to-production gap.** Models that work in a notebook were never engineered to run reliably, be served, monitored, or audited.
- **A widening divide** between fast-moving data-science experiments and the slower discipline of production engineering.
- **Ungoverned generative AI.** GenAI is adopted under pressure — but without a way to tell whether it actually helps, or what it costs.

> ▲ **The real GenAI risk**
>
> It is not that generative AI does nothing — it is that it produces *plausible* output that no one has verified, at a cost no one is tracking.

The result is a recognizable pattern: stalled pilots, sunk cost, and a quiet erosion of confidence in the whole programme.

The Firefly answer

Firefly DataScience turns this operating model around — it treats data science as **production engineering from day one**, and generative AI as a **governed accelerator** rather than an oracle. It does three things together: it **automates** the choice and tuning of models (AutoML); it lets an AI **agent iterate** on the solution under a strict, budgeted loop; and it wraps every generative step in a **measurable gate** — nothing is adopted unless it is proven to improve the result on the organization's own data.

- Proven, classical machine learning does the heavy lifting — cheap, fast, and reproducible.
- Generative AI proposes ideas; a deterministic engine decides which ones earn their place.
- Everything is open and swappable — no lock-in to a vendor, cloud, or single library.
- Serving, data validation, lineage and benchmarking are built in, not bolted on.

Part I · Strategy · Chapter 3

Five outcomes for the business

Stripped of the engineering detail, Firefly DataScience changes five things that leaders care about.

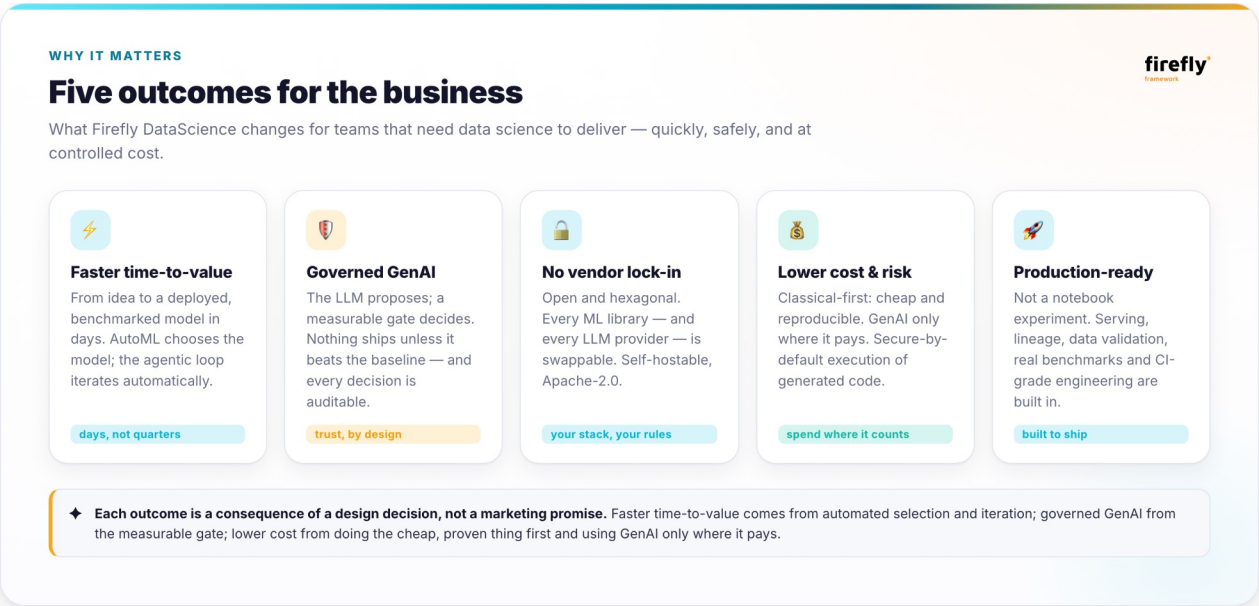


Figure 2. *The five business outcomes Firefly DataScience is designed to deliver.*

Each outcome is a direct consequence of a design decision, not a marketing promise. Faster time-to-value comes from automating model selection and iteration. Governed GenAI comes from the measurable gate. The absence of lock-in comes from an open, hexagonal architecture. Lower cost and risk come from doing the cheap, proven thing first and using generative AI only where it pays. And production-readiness comes from treating serving, validation and lineage as first-class from the start.

Part I · Strategy · Chapter 4

Governed GenAI you can trust

Generative AI is the accelerator — but it is kept on a governed leash.

The single most important idea in the framework is also the easiest to explain. Generative AI is fast and creative, but it cannot be trusted to be *right*. So it is never allowed to decide. It proposes; the engine measures; a gate keeps only what is proven to help.

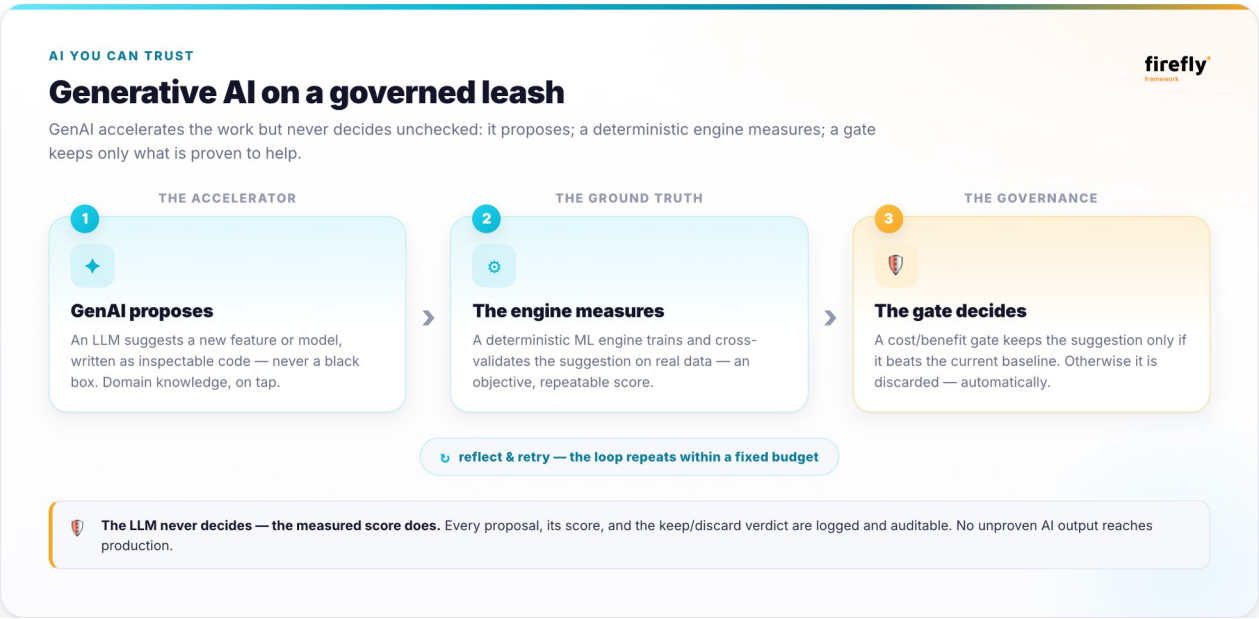


Figure 3. *How generative AI is governed: propose, measure, then keep only what beats the baseline.*

✓ Auditable by construction

Every proposal, the score it earned, and the keep-or-discard verdict are recorded. A risk, compliance, or audit function can see exactly why the system did what it did — and no unproven AI output ever reaches production.

Part I · Strategy · Chapter 5

From data to production value

The same pipeline carries a dataset all the way from raw to served — and keeps verifying as it goes.

There is one path, not a hand-off between disconnected tools. Data is validated, a model is selected automatically, generative AI adds value where it earns it, an agent iterates within a budget, and the winner is served with full lineage. What used to take quarters of bespoke work compresses into days.

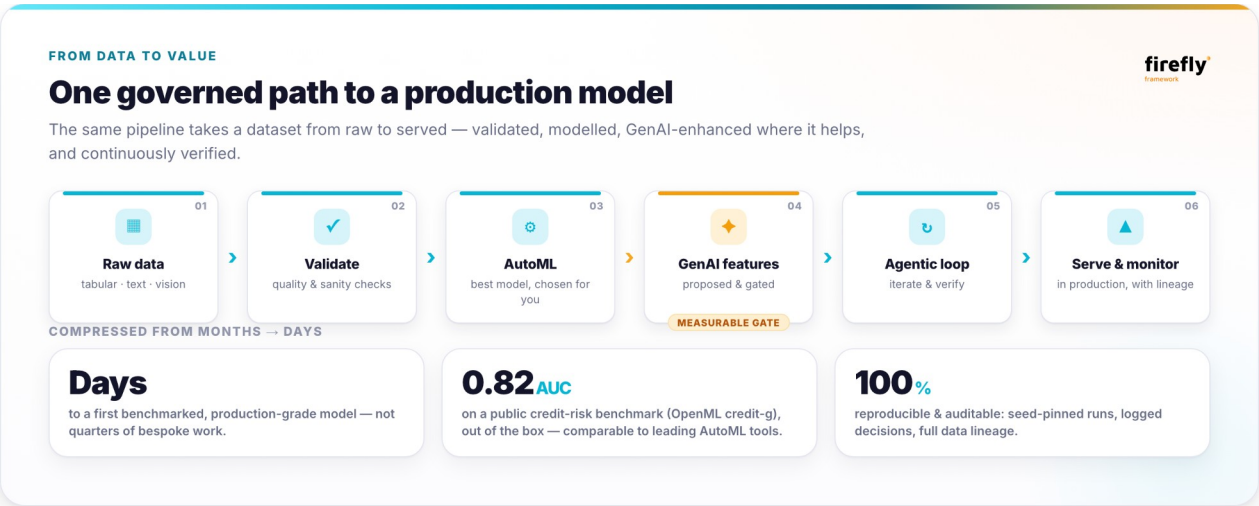


Figure 4. *One governed path from raw data to a served, monitored model — with representative results.*

The numbers are grounded. On the public OpenML `credit-g` credit-risk benchmark, the framework's automated pipeline reaches **0.82 ROC-AUC out of the box** — comparable to leading AutoML tools, with no manual tuning, and fully reproducible from a fixed seed. Where the data has structure a linear model cannot reach, gated GenAI feature engineering adds a further, measured lift — and where it would not help, the gate simply declines it.

Part I · Strategy · Chapter 6

How it's different, and how to start

Most teams reach for bespoke notebooks or a closed AutoML product. Firefly DataScience changes the trade-offs on both.

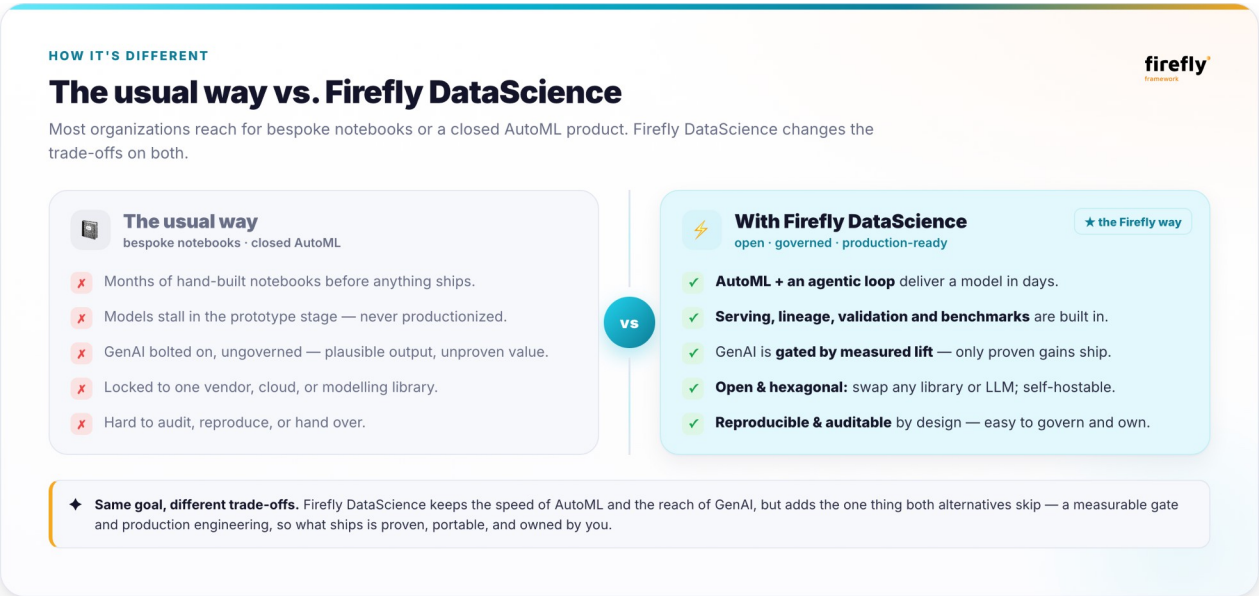


Figure 5. *The usual way of working versus Firefly DataScience.*

It fits wherever an organization needs to apply data science quickly but cannot accept a black box: regulated industries, internal platform teams, and consultancies who must hand over something a client can own, audit, and run themselves. Because it is open and self-hostable, it sits comfortably inside existing cloud, data and security boundaries.

A leader's concern	How Firefly DataScience answers it
“Will this actually ship?”	Serving, lineage and benchmarks are built in — production is the default, not an afterthought.
“Can we trust the AI?”	GenAI is gated by measured lift and fully logged; nothing unproven reaches production.
“Are we locked in?”	Open, Apache-2.0, hexagonal — every library and LLM is swappable; self-hostable anywhere.
“What will it cost?”	Classical-first by default; generative AI is used only where it demonstrably pays.

Getting started

Adoption is deliberately low-risk: it is open-source, installs with one command, and proves itself on a single dataset. A typical first engagement is a short, contained pilot:

1. **Pick one decision** that a model could improve, and a dataset that describes it.
2. **Run the automated pipeline** to get a benchmarked baseline model in days.
3. **Turn on governed GenAI** to see whether it lifts the result — measured, not assumed.
4. **Review the evidence** — the leaderboard, the gate's decisions, the lineage — with the business and risk owners.
5. **Decide to scale** on the strength of a working, auditable result rather than a slide.

> ⓘ **Where it stands today**
>
> Firefly DataScience is a working, openly-developed framework (Apache-2.0) with a green continuous-integration pipeline and results validated on public benchmarks. It is young — best suited to teams that want to own and shape their AI stack, rather than a mature, off-the-shelf product with a catalogue of customer case studies. That is precisely where a strategic early adopter captures the most value.

It is built on the broader **Firefly Framework** and its agentic runtime, so an investment here compounds across a coherent, open ecosystem rather than a single tool.

PART II

Architecture & concepts

How the framework is built — hexagonal ports and adapters, the classical-first engine, the gated GenAI accelerator, the agentic loop, and the security model.

Part II · Architecture · Chapter 7

The big picture: hexagonal, ports & DI

Firefly DataScience is a hexagonal, auto-configured framework: a lean DI container wires ports to adapters, and a Spring-Boot-style application context boots it all from packaging entry points.

The design goal is single-minded — the domain never depends on a vendor SDK, and an adapter can be added, swapped, or overridden without touching calling code. The core stays importable with **no** optional ML extra installed; vendor imports live inside adapters and `@bean` methods, never at module top level. Everything below is how that discipline is enforced in code.

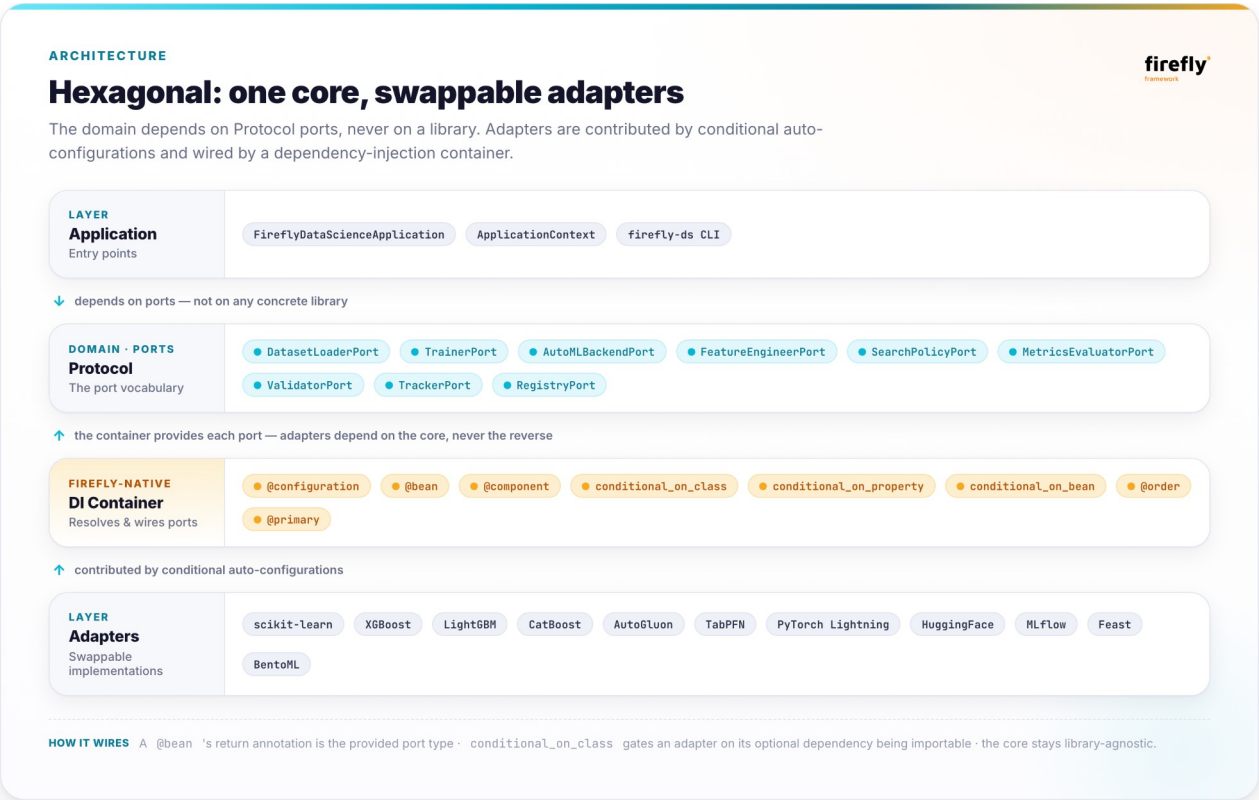


Figure 6. *Hexagonal architecture: a library-agnostic core, swappable adapters.*

The five layers

The framework is organised top-to-bottom so the domain never reaches a vendor SDK. Five layers, each with one job:

6. **Application** — `FireflyDataScienceApplication` / `ApplicationContext`: the bootstrap and the started, wired context you resolve beans from.
7. **Auto-configuration** — `@auto_configuration` `@configuration` classes discovered via entry points; each contributes adapters conditionally.
8. **Container** — the `Container`: a type-hint-driven IoC container with singleton/transient scopes and constructor injection.

9. **Domain / Ports** — `Protocol` interfaces plus the light, dependency-free core types in `core.types` (`TaskType`, `Modality`, `Scope`).
10. **Adapters** — concrete implementations of the ports backed by optional extras (scikit-learn, OpenML, deep-learning, GenAI, …), each gated by a condition.

> ⓘ **The core stays light**
>
> `core.types` enforces the rule with hand-written `StrEnum`s (`TaskType`, `Modality`, `Scope`) and no third-party ML imports. A `from fireflyframework_datascience.automl import AutoML` is cheap because scikit-learn is only loaded when you actually call `fit`.

Ports live in their own domain modules

A **port** is a `Protocol` the domain depends on; an **adapter** is a concrete class that implements it. The container binds them by type annotation, so swapping an adapter never touches calling code. Crucially, each port is declared in **its own domain module** — there is no central `ports` package:

Port	Declared in	What it contracts
`DatasetLoaderPort`	`datasets`	Load a dataset by name.
`TrainerPort`	`models`	Build and fit an estimator for a task.
`AutoMLBackendPort`	`automl`	Run the classical fit loop.
`FeatureEngineerPort`	`features`	Engineer features (incl. gated GenAI).
`SearchPolicyPort`	`search`	Optimize the CV objective over a space.
`MetricsEvaluatorPort`	`evaluation`	Score CV and held-out metrics.
`ValidatorPort`	`validation`	Validate a dataset before training.
`TrackerPort` / `RegistryPort`	`tracking`	Log runs; persist/retrieve models.

Each is a contract the domain calls; the concrete class that fulfils it is decided at boot. An auto-configuration contributes the adapter, gated on the optional dependency being importable. Note the load-bearing detail in the `@bean` method: **its return annotation is the provided type**, so the container registers the adapter under the port.

```
from fireflyframework_datascience.container.conditions import (
    auto_configuration,
    conditional_on_class,
)
from fireflyframework_datascience.container.stereotypes import bean, configuration
from fireflyframework_datascience.datasets import DatasetLoaderPort


@auto_configuration
@conditional_on_class("sklearn")
@configuration
class DatasetsAutoConfiguration:
    @bean(name="sklearn_dataset_loader")
    def sklearn_loader(self) -> DatasetLoaderPort:        # return annotation ==
provided type
        from fireflyframework_datascience.datasets.adapters import
SklearnDatasetLoader
        return SklearnDatasetLoader()
```

At boot, `_apply_one` reads `get_type_hints(method)["return"]` to decide what the bean is registered as; a `@bean` method with no return annotation is simply skipped. Resolving `DatasetLoaderPort` then yields whichever adapter won.

The DI container

The `Container` is a lean IoC container — resolution is by type annotation, with constructor injection and circular-dependency detection. A small, stable vocabulary spans the wiring layer:

- `@configuration` / `@bean` — mark a class as holding factory methods; mark a method as a bean factory. `@bean` defaults to `scope=Scope.SINGLETON` and `primary=False`.
- `@component` — mark a class as injectable by its own type.
- `@auto_configuration` — mark a class discoverable via the entry-point group.
- `@order` — set ordering (lower runs/resolves first); `@primary` breaks ambiguity when several beans share a type.
- `Scope` — `SINGLETON` (cached, the default) or `TRANSIENT` (a new instance each resolve).

Conditions gate both whole auto-configurations and individual `@bean` methods: `conditional_on_class("sklearn")` matches when the extra is importable, `conditional_on_property("genai.enabled")` reads a dotted path off the config, and `conditional_on_bean` / `conditional_on_missing_bean` query the partially-wired container. The last is the **secure-by-default override hook** — a framework default applies only when you have not already registered your own.

Always-on core, then everything else

Adapter packages register their auto-configuration class under the `firefly_datascience.auto_configuration` entry-point group. At startup

`discover_auto_configurations()` loads every class in the group, tolerating any whose optional extra is missing, then sorts by `@order`. `CoreAutoConfiguration` is the always-on reference example: it has no `@conditional_on_class`, so it always applies, registering a single `RuntimeInfo` bean that snapshots the framework version, Python version, platform, default ML framework, and whether GenAI is enabled.

The application lifecycle

`FireflyDataScienceApplication.start()` runs a fixed sequence, mirroring pyfly's lifecycle: load config → print the banner → create the `Container` and register the config as a bean → discover auto-configurations (entry points + any extras), de-duplicate, sort by `@order` → evaluate each one's conditions and register the surviving `@bean` methods → `eager_init()` all singletons (fail-fast) → return a ready `ApplicationContext`.

```
from fireflyframework_datascience.application import FireflyDataScienceApplication

# One-call bootstrap.
ctx = FireflyDataScienceApplication.run()

print(ctx.bean_count, "beans")
print([ac.__name__ for ac in ctx.applied_auto_configurations])

loader = ctx.get(DatasetLoaderPort)           # resolve a bean by type
tracker = ctx.get_optional(SomeOptionalPort)  # None if not wired
```

You can steer the boot without forking: `extra_auto_configurations=[...]` **appends** your own to whatever was discovered, while `auto_configurations=[...]` **replaces** discovery entirely (handy for hermetic tests). Pass `print_output=False` to silence the banner and wiring summary.

✓ Expected

When the banner is on, boot ends by printing a wiring summary whose line *shape* is fixed — `profiles`, `beans`, `auto-config`, `ml framework`, `genai`, `sandbox`. The exact bean count and applied list depend on which optional extras are installed; for example, *3 applied (CoreAutoConfiguration, DatasetsAutoConfiguration, ModelsAutoConfiguration)*, *ml framework: sklearn*, *genai: disabled*.

◆ Why the architecture is the governance

The same separation that keeps vendor SDKs out of the domain keeps GenAI out of the decision path. GenAI lives in **adapters** behind ports; the deterministic classical engine trains, scores and selects. The container resolves a port — and the classical engine decides whether a proposal survives a measured improvement over a seeded baseline. **The LLM proposes; the classical engine decides** is not a slogan here, it is enforced by the wiring.

Part II · Architecture · Chapter 8

Classical-first AutoML

Cross-validate a panel of tabular models, tune the contenders, and fit the winner — in three lines.

The `AutoML` engine is the front door of Firefly DataScience. It validates a dataset, cross-validates every trainer that supports the task (optionally tuning each one), and returns a fitted winner together with the full leaderboard. It is import-light: scikit-learn is only loaded when you actually call `fit`.

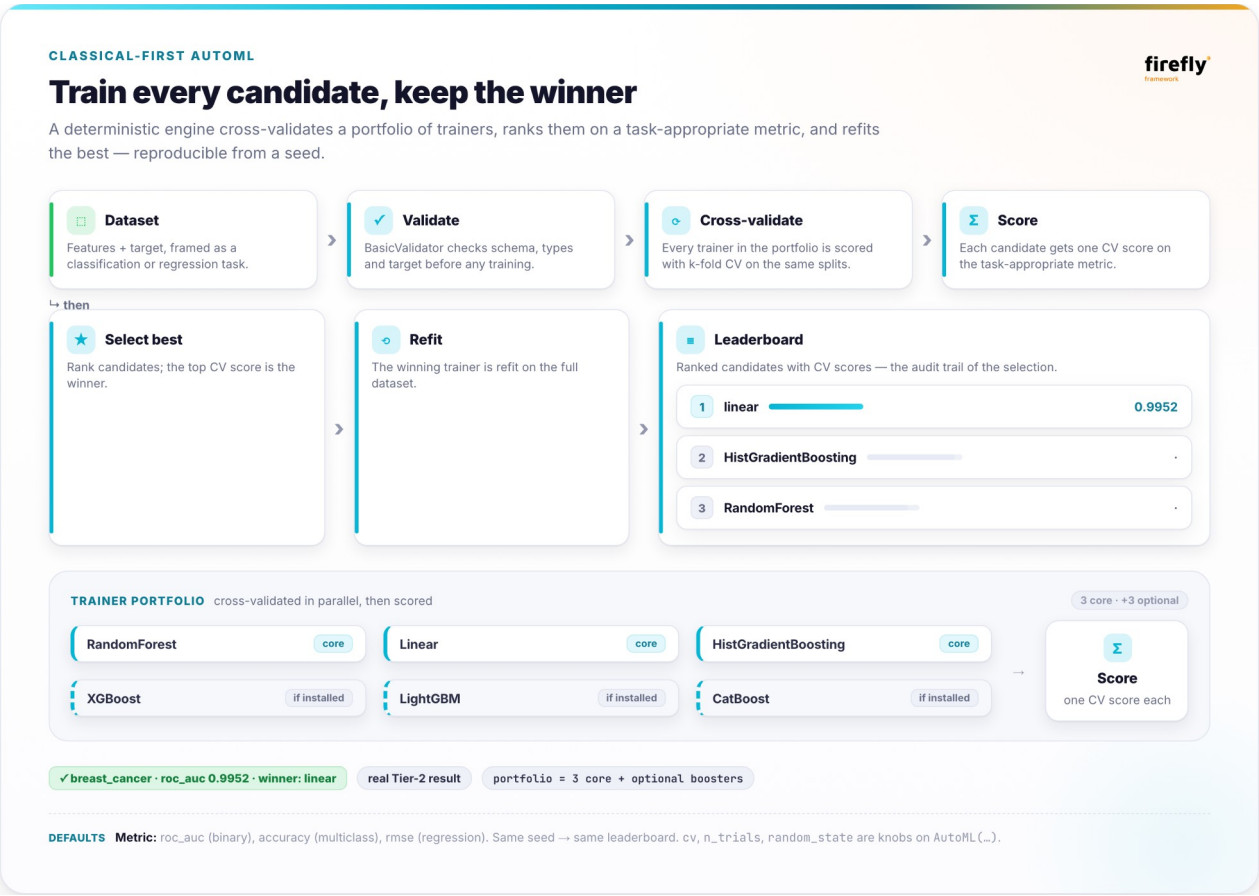


Figure 7. *The classical-first AutoML loop: cross-validate, score, select, refit.*

Quick start

Construct the engine, call `fit`, and read the result. The task and metric are inferred from the dataset unless you override them.

```
from fireflyframework_datascience.automl import AutoML
from fireflyframework_datascience.datasets.adapters import SklearnDatasetLoader

dataset = SklearnDatasetLoader().load("breast_cancer")
train, test = dataset.train_test_split(test_size=0.25)

result = AutoML(cv=5, n_trials=20, random_state=42).fit(train)

print(result.best_model.name)         # the winning trainer
print(result.leaderboard_table())     # one line per candidate, best CV score first
print(result.evaluate(test))          # EvaluationResult on held-out data
```

The constructor is keyword-only for its knobs (`cv=5`, `n_trials=20`, `random_state=42` by default). Anything left as `None` falls back to sensible defaults: trainers `[RandomForestTrainer(), LinearTrainer(), HistGradientBoostingTrainer()]` (plus `XGBoostTrainer`, `LightGBMTrainer`, `CatBoostTrainer` when those libraries are installed), the `SklearnMetricsEvaluator`, and the `DefaultSearchPolicy`. In an application, `AutoML.from_context(app, ...)` wires the components from the container instead.

The fit loop

For each trainer that `supports(task)`, the engine runs **validate → cross-validate → score → select → refit → leaderboard**:

11. Build the trainer's hyperparameter search space — but only when `n_trials > 1`; with `n_trials <= 1` the space is empty and the search collapses to a single default-hyperparameter evaluation.
12. Run the search policy, whose objective wraps the estimator in a preprocessing pipeline and cross-validates it (`cross_val_score`, `cv` folds).
13. Record a `LeaderboardEntry` with the best CV score. A candidate that raises during CV is logged and scored `-inf`, so one broken estimator never aborts the run.
14. After all candidates are scored, refit the highest-scoring trainer on the full training data and wrap it as a `Model`.

Selection is strictly by CV score — greater is always better. The preprocessing pipeline is built automatically from the column dtypes (median impute + `StandardScaler` for numerics, most-frequent impute + `OneHotEncoder` for categoricals), so the same pipeline serves trees and linear models alike.

> ⓘ **Two scores, one winner**
>
> `result.metric` is the human-facing metric name (e.g. `roc_auc`); `result.cv_scoring` is the sklearn scoring string actually used for CV. Default metrics are `roc_auc` (binary), `accuracy` (multiclass), and `rmse` (regression). Because loss-style metrics map to negated scorers, CV scores are always maximized.

Reading the leaderboard

`fit` returns an `AutoMLResult` carrying the fitted winner, the sorted leaderboard, and the evaluator used. The headline attributes are `best_model`, `best_score`, `leaderboard`, `metric`, and `task`, plus `predict`, `predict_proba`, and `evaluate` for using the winner.

```
result.best_model          # Model: name, estimator, task, feature_names, params
result.best_score          # leaderboard[0].cv_score (the top CV score)
result.leaderboard         # list[LeaderboardEntry], sorted best-first

result.predict(test.X)               # winner predictions
result.predict_proba(test.X)         # class probabilities (classification)
eval_result = result.evaluate(test)  # full metric panel on held-out data
print(eval_result.primary_metric, eval_result.primary_value)

print(result.leaderboard_table())    # one line per candidate
```

Each `LeaderboardEntry` holds `model_name`, `params`, `cv_score`, and `metric`, and stringifies as a tidy `"{model:<24} {metric}={score:.4f}"` line — the trainer name left-padded to 24 columns, the score to four decimals. `best_score` is a property reading the top entry, so it always agrees with the first row of the table.

✓ Expected

On the breast-cancer quick start, automated selection reaches **roc_auc ≈ 0.9952**, and the **linear** model wins — a logistic regression, on properly scaled features, edges out the trees on this dataset. That is the point of running the panel: the engine selects by measured CV score, not by reputation. Values vary slightly by environment and library versions.

◆ Classical owns the decision

`AutoML` is pure classical machine learning — deterministic, seeded, and reproducible. Where GenAI enters elsewhere it only ever *proposes*; this engine *decides* by cross-validated score. The search is owned by Optuna and scikit-learn, never by a language model.

Calibration, ensembling, and the right metric

Single-best selection by accuracy is rarely the production answer. Three keyword switches turn the panel into a production-grade selector — each off by default, each composable:

- **Calibrate** (`calibrate=True`) wraps the winner in a cross-fit `CalibratedClassifierCV`, so its predicted probabilities are trustworthy — essential for risk- and cost-sensitive decisions. The evaluator then reports the **Brier score** (lower is better-calibrated) alongside the usual panel.
- **Ensemble** (`ensemble=True`) stacks the top-k candidates behind a cross-fit meta-learner (`StackingClassifier` / `StackingRegressor`) — the standard last-mile lift over single-best.

- **Select on the right metric** — pass `metric="average_precision"` to choose the winner on **PR-AUC**, the correct target for imbalanced binary problems where ROC-AUC over-credits.

```
from sklearn.model_selection import StratifiedKFold, TimeSeriesSplit

# A calibrated stacking ensemble, selected on PR-AUC, with an explicit CV strategy.
result = AutoML(
    cv=StratifiedKFold(5, shuffle=True, random_state=42),  # or TimeSeriesSplit for
temporal data
    calibrate=True,
    ensemble=True, ensemble_size=3,
).fit(train, metric="average_precision")

print(result.best_model.name)                          # "stacking_ensemble"
print(result.evaluate(test).metrics["brier_score"])    # probability quality (lower
is better)
```

◆ **A splitter, not just a fold count — no silent leakage**

`cv` accepts any scikit-learn splitter. Use `TimeSeriesSplit` for temporal data (forward-chaining, so the model never trains on the future) or `GroupKFold` for grouped data (a group never spans train and test). The same splitter drives every candidate, so the leaderboard stays comparable.

Explaining the winner

A model you cannot explain is a model you cannot ship into a regulated domain. Every `AutoMLResult` can explain its winner with **deterministic, peer-reviewed** methods — never an LLM. `explain()` returns global feature importances; with the optional `explain` extra, SHAP adds per-prediction (local) attributions for "why did *this* case score the way it did?".

```
explanation = result.explain(test)         # GlobalExplanation
print(explanation.method)                   # "permutation_importance" (or "shap")
for name, importance in explanation.top(8):
    print(f"{name:<26} {importance:+.4f}")
```

ⓘ **Classical proposes, classical decides — and classical explains**

Importances come from permutation importance (the dependency-free default) or SHAP, both reproducible from a seed. Just as GenAI only *proposes* and the classical engine *decides*, here the classical engine also *explains* — the language model is never in the explanation path.

Part II · Architecture · Chapter 9

GenAI as a gated accelerator

The LLM proposes feature code; classical cross-validation decides what survives.

Firefly DataScience treats generative AI as a *proposer*, never a judge. A language model suggests pandas snippets that add new columns; a deterministic engine measures the cross-validation lift of each one, and a `CostBenefitGate` keeps a feature only if it beats the current baseline by a measurable margin. The LLM never touches the score.

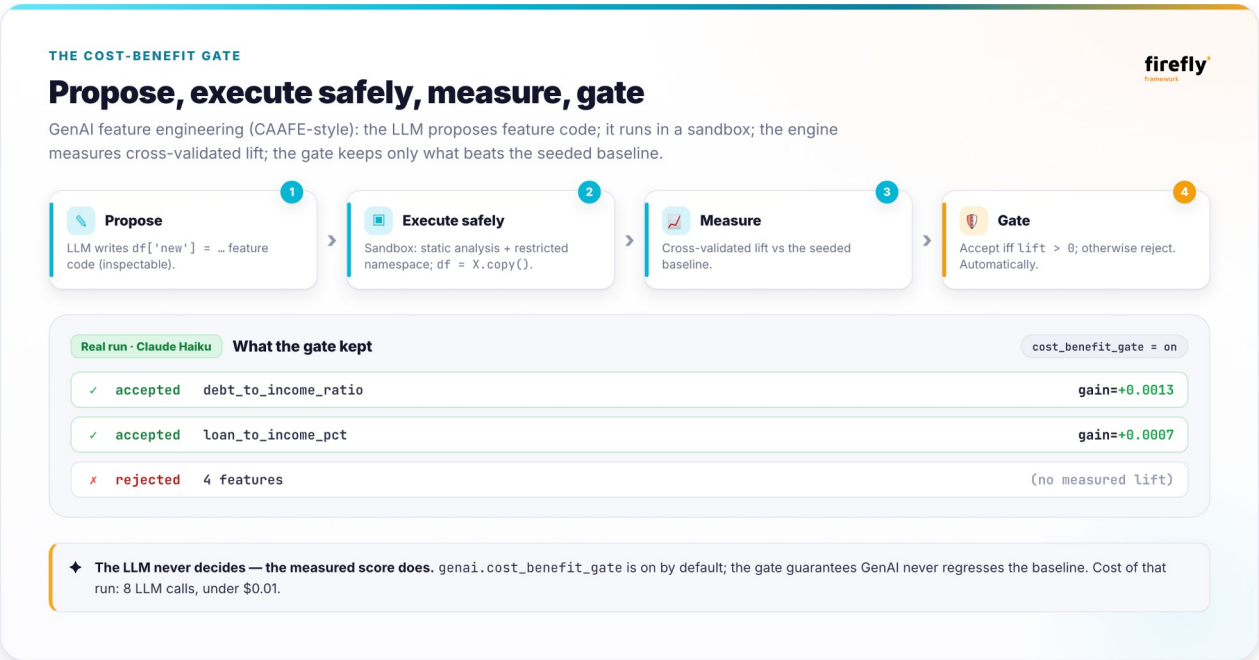


Figure 8. *GenAI feature engineering: propose, execute safely, measure, gate.*

The loop

`GenAIFeatureEngineer` runs **propose → execute → measure → gate**:

15. **Propose** — a `FeatureProposer` returns a list of `FeatureProposal`s (`name`, `code`, `rationale`).
16. **Execute** — `FeatureCodeExecutor` statically vets and safely runs each snippet against a copy of the frame.
17. **Measure** — a classical estimator scores the candidate frame via cross-validation.
18. **Gate** — `CostBenefitGate` accepts the feature only if the score improves by more than `min_gain`.

Each accepted feature is folded into the working frame, so the next proposal is measured against the *improved* baseline — features must earn their keep on top of everything kept so far. Two proposers satisfy the `FeatureProposer` protocol: `StaticFeatureProposer` (a fixed, offline set of features — ideal for tests and codifying domain knowledge) and `AgentFeatureProposer` (a real LLM, built lazily on first use).

Running it

Drive the loop with a deterministic proposer — no LLM required — and read the audit trail off the result.

```
from fireflyframework_datascience.features import FeatureProposal,
StaticFeatureProposer
from fireflyframework_datascience.features.genai import GenAIFeatureEngineer

proposer = StaticFeatureProposer([
    FeatureProposal(
        name="debt_to_income_ratio",
        code="df['debt_to_income_ratio'] = df['loan_amount'] / (df['income'] + 1)",
        rationale="The withheld latent driver of credit risk.",
    ),
    FeatureProposal(
        name="loan_to_income_pct",
        code="df['loan_to_income_pct'] = (df['loan_amount'] / df['income']) * 100",
        rationale="Loan size relative to income.",
    ),
])

engineer = GenAIFeatureEngineer(proposer, cv=5, max_features=5)
result = engineer.engineer(train)
print(result.summary())
```

`engineer()` returns an `EngineeringResult` with the engineered `dataset`, the `baseline_score`, the `final_score`, the `lift`, and full lists of `accepted` and `rejected` features. An `AcceptedFeature` carries the proposal, its `score`, and the `gain` over the previous best; a `RejectedFeature` carries the proposal, a `reason`, and the candidate `score`. A proposal is rejected when its code is unsafe, fails at runtime, adds no new numeric column, or produces no measured lift — in which case the reason reads `no lift (<candidate> <= <current>)`.

A real-LLM result

With a real model (`anthropic:claude-haiku-4-5`), the engineer was asked to improve a synthetic credit-risk dataset whose risk is driven by *debt-to-income* — a ratio deliberately withheld from the model. Claude proposed six features from the schema alone; the gate **accepted** the two that lifted a logistic baseline and **rejected** the four that did not, for under \$0.01:

Verdict	Feature	Gain
accepted	`debt_to_income_ratio`	+0.0013
accepted	`loan_to_income_pct`	+0.0007
rejected	`employment_stability_score`	no measured lift
rejected	`prior_default_flag`	no measured lift
rejected	`default_frequency`	no measured lift

Verdict	Feature	Gain
rejected	`income_loan_buffer`	no measured lift

The LLM discovered the latent driver from the schema alone — and the gate kept only what was proven on the data. The cost-benefit gate is on by default (`genai.cost_benefit_gate = True`); raise `min_gain` to demand a feature clear a meaningful bar before it earns its complexity.

> ◆ **The LLM never decides — the measured score does**
>
> `CostBenefitGate` is the governance primitive that keeps GenAI honest: `gate.accepts(current, candidate)` is `True` only when `(candidate - current) > min_gain`. A proposal that does not measurably beat the seeded classical baseline is rejected. The model proposes; the data decides what survives.

Part II · Architecture · Chapter 10

The agentic ML-engineering loop

An LLM proposes, the classical engine decides — a deterministic verifier, not "it ran", is the judge.

The agentic loop realizes the SOTA AutoML pattern grounded on a deterministic executor: an LLM *proposes* a solution (a trainer plus hyperparameters), the classical engine *trains and cross-validates* it, and a **Verifier** — a stage distinct from execution-success — decides whether the result is genuinely good. Search is greedy with reflection over the attempt history, bounded by an iteration and patience budget.

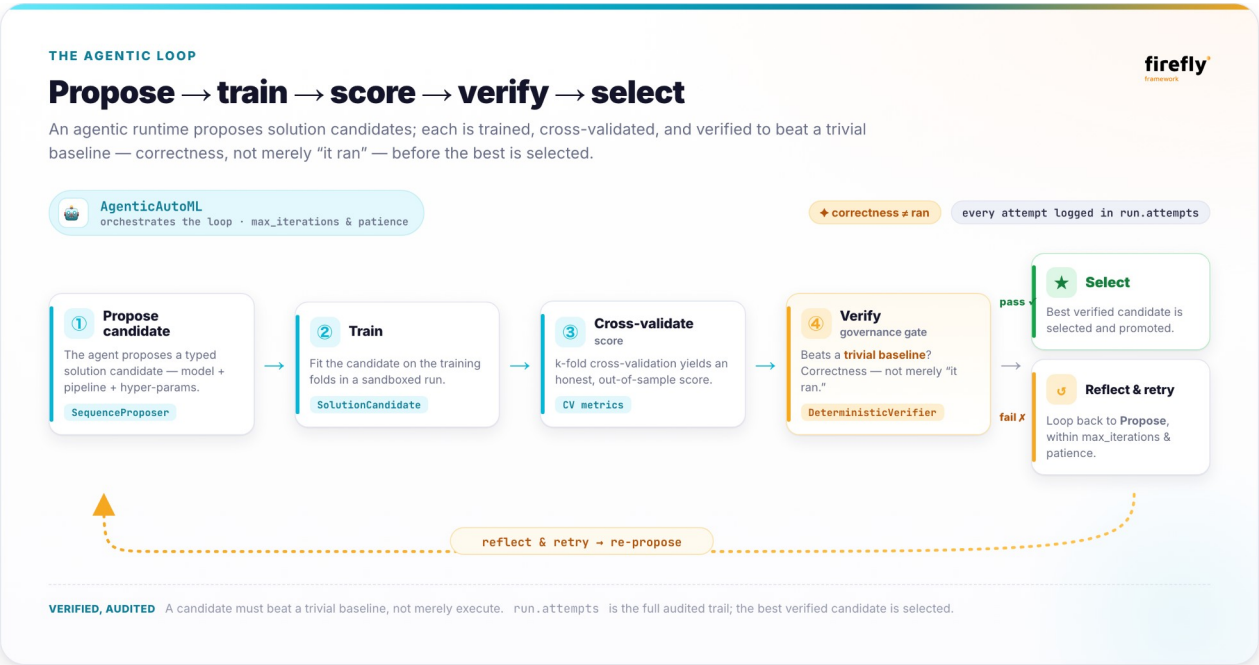


Figure 9. *The agentic loop: propose, train, score, verify, select — reflect and retry.*

The pieces

- `SolutionCandidate` — a proposal: `trainer`, `params`, `rationale`.
- `AgenticAutoML` — the loop engine that runs **propose → train/CV → verify → reflect → select**.
- `DeterministicVerifier` — the correctness check: a finite score that beats the trivial baseline by a `margin`.
- `SequenceProposer` — a deterministic proposer for tests and fixed strategies; `AgentSolutionProposer` is the LLM-backed sibling.
- `EngineeringRun` — the full, auditable trace plus the refit best model.

Running it

```
from fireflyframework_datascience.engineering import SolutionCandidate,
SequenceProposer
from fireflyframework_datascience.engineering.loop import AgenticAutoML

proposer = SequenceProposer([
    SolutionCandidate("linear", rationale="cheap baseline"),
    SolutionCandidate("random_forest", {"n_estimators": 300}),
    SolutionCandidate("hist_gradient_boosting", {"learning_rate": 0.05}),
])

loop = AgenticAutoML(proposer, max_iterations=8, patience=3, cv=5)
run = loop.solve(train)

print(run.summary())
```

✓ Expected

A run prints a one-line recap: *Agentic AutoML: 3 attempts (2 verified); best=hist_gradient_boosting roc_auc=0.9123 (baseline 0.5000)*. The engine seeds candidates from `propose_initial`, then repeatedly calls `propose_next` until a candidate is `None`, the iteration budget is spent, or patience runs out.

Correctness is not "it ran"

A run that produces a number is not the same as a run that produced a *good* number. Verification is a separate stage: each candidate is trained, cross-validated, and then **verified** to beat a trivial baseline (a `DummyClassifier` with `strategy="prior"` / `DummyRegressor` with `strategy="mean"`) by a `margin`. Only a `valid` verdict that improves on the running best can take the lead. Verdicts read like a review:

```
Verdict(valid=False, reason="training failed or produced a non-finite score", score=-
inf)
Verdict(valid=False, reason="does not beat the trivial baseline (0.5010 <= 0.5000)",
score=0.5010)
Verdict(valid=True,  reason="beats trivial baseline by +0.4123", score=0.9123)
```

The full trail is auditable: `run.attempts` is every iteration in order — each with its candidate, score, and verdict — and `run.valid_attempts`, `run.best_candidate`, `run.best_score`, and `run.model` are derived from it. Two knobs bound the search: `max_iterations` (default 8, the cap on reflection rounds after seeding) and `patience` (default 3, consecutive non-improving attempts allowed before early stopping). An improving verified attempt resets patience to the full budget.

◆ The LLM proposes; a deterministic verifier rules

The LLM never gets the last word. It *suggests* the next `(trainer, params)` candidate; the classical engine cross-validates it and `DeterministicVerifier` rules on whether it actually beats the baseline. A candidate that runs but fails to clear the baseline is rejected — so the loop can only ever return a model that measurably earned its place.

Part II · Architecture · Chapter 11

Deep learning, serving & lineage

Neural networks, transformers and CNNs widen *what* can be proposed — they do not change *who* decides.

Deep learning lives behind the **same ports** as the classical adapters. The `dl` module defines `DLTrainerPort` for neural trainers and `TabFMPort` for tabular foundation models; both share the exact shape the classical engine expects — `name`, `supports(task)`, and `fit(dataset) -> Model`. That parity is the point: a deep-learning adapter is not a special case, so AutoML can rank an `MLPTrainer` against a gradient-boosted tree with no extra wiring. A verified `MLPTrainer` (scikit-learn) ships as the dependency-light reference; the heavy adapters (`TorchTabularTrainer` for PyTorch Lightning / HuggingFace, `TabPFNPredictor`) are gated behind extras and raise a clear `AdapterUnavailableError` when those extras are missing.

Modalities at a glance

The framework names five modalities in the `Modality` enum. Three have shipping adapters today; two are reserved in the type system but have no built-in adapter yet — you would supply your own against the relevant port on the same `supports` + `fit` contract.

Modality	Port	Reference adapter	Status
`TABULAR`	`DLTrainerPort` / `TabFMPort`	`MLPTrainer`, `TabPFNPredictor`, `TorchTabularTrainer`	shipping
`TEXT`	`TextClassifierPort`	`HFTextClassifier` (DistilBERT)	shipping
`VISION`	`ImageClassifierPort`	`TorchCNNClassifier`	shipping
`TIMESERIES`	—	none built in	reserved
`MULTIMODAL`	—	none built in	reserved

> ⓘ **Reserved, not implemented**
>
> `TIMESERIES` and `MULTIMODAL` exist in the `Modality` enum (and `TaskType.FORECASTING` exists for forecasting tasks), but the published package ships no adapter for them. The enum names the design space; bring your own adapter on the same `supports` + `fit` contract and it satisfies the port like any other.

Serving the winner

A fitted `Model` is served by a `ModelServerPort`. The container registers `LocalModelServer` as the in-process default — no external dependency — and a `BentoMLModelServer` is available behind the `serving`

extra for packaging and deployment. Because both expose the same `name` / `load` / `predict` surface, swapping is a one-line change.

```
from fireflyframework_datascience.serving import LocalModelServer

server = LocalModelServer()        # server.name == "local"
server.load(result.best_model)     # hand it the trained winner
preds = server.predict(X_new)
```

> ▲ **Load only from trusted locations**
>
> A `Model` persists with `joblib`, which uses `pickle` — and `pickle` executes arbitrary code on load. Calling `predict` before `load` raises `FireflyDataScienceError("No model loaded — call load(model) first")`. Treat the registry as the trusted source for `Model.load`; never deserialize an artifact from untrusted input.

Tracking, registry & lineage

The same pattern — a narrow port, a zero-dependency default, an opt-in adapter behind an extra — extends to operations. A `TrackerPort` records params, metrics, and model artifacts (`NoOpTracker` by default, `MLflowTracker` when `tracking_enabled` is `True`); a separate `RegistryPort` persists and retrieves models by name and version; and a `LineagePort` emits a `LineageEvent` with input/output dataset references (`NoOpLineage` by default, `OpenLineageEmitter` behind the `lineage` extra). Moving from the in-process default to MLflow, BentoML, or OpenLineage is a configuration change, not a rewrite.

> ◆ **The same gate, applied to operations**
>
> A heavyweight neural trainer earns its place only when it beats the dependency-light baseline on the metric — the cost-benefit gate, not the adapter, makes the call. The defaults are deterministic and dependency-free, so a run that scored well in development behaves identically when served; heavier backends are opt-in, adopted when they earn their keep.

Part II · Architecture · Chapter 12

The security model

Firefly DataScience treats LLM-generated code as hostile input: statically vetted, run with a stripped namespace, and — for untrusted data — pushed behind a sandbox or a human.

> **▲ GenAI is off until you enable it**
>
> Firefly is classical-first. `genai.enabled` defaults to **`False`**, so none of the code-generating accelerators run unless you opt in. Until then there is no LLM, no generated code, and no executor invocation — the secure default is *nothing executes*. Everything below describes the controls that engage once you turn GenAI on.

The GenAI accelerators ask a model to *write Python that runs against your data*. That is an attack surface. The model is **not** trusted: it might emit code that exfiltrates data or escalates privilege, the data itself might carry a prompt injection, or the model might hallucinate code that corrupts the frame. We *do* trust the host process, the installed libraries, and the configuration. The goal: a wrong or hostile snippet cannot do more than fail loudly. The defence is three layers deep.

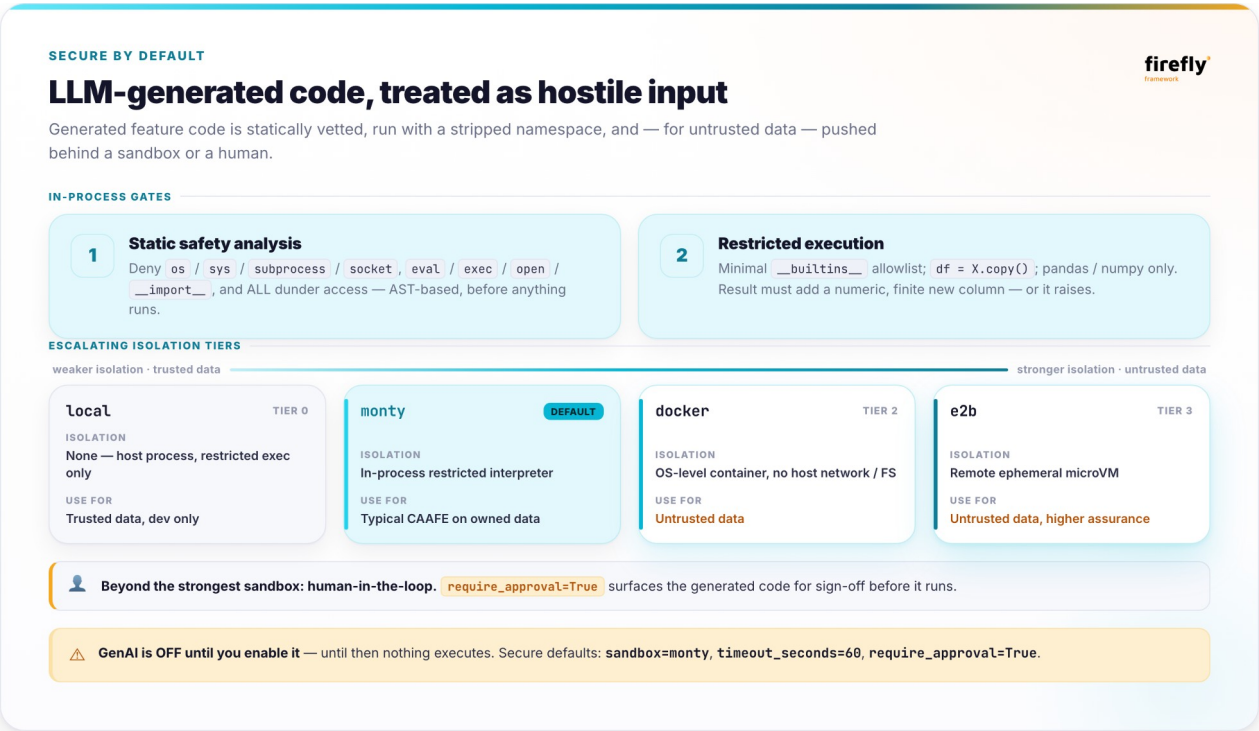


Figure 10. *Secure-by-default: static analysis, restricted execution, tiered sandbox.*

Layer 1 — static safety analysis

Before a single byte of model output executes, `FeatureCodeExecutor` runs it through the AST-based static analyzer reused from `fireflyframework_agentic.execution`. The `SafetyPolicy` denies dangerous modules, dangerous builtins, and **all dunder access** — which is how sandbox escapes are typically built:

```
from fireflyframework_agentic.execution import SafetyPolicy

policy = SafetyPolicy(
    denied_modules=frozenset(
        {"os", "sys", "subprocess", "shutil", "socket", "pathlib", "importlib",
"builtins"}
    ),
    denied_builtins=frozenset(
        {"eval", "exec", "compile", "open", "__import__",
         "input", "globals", "locals", "vars", "getattr", "setattr"}
    ),
    deny_dunder_access=True,  # blocks ().__class__.__bases__... escapes
)
```

The executor constructs exactly this policy in its `__init__`, calls `analyze_code(code, policy)`, and refuses to proceed unless `report.safe` is true — surfacing each `report.violations[*].message` in the raised error. A rejected snippet raises a typed `FeatureExecutionError` (a subclass of `FireflyDataScienceError`): it does not run, does not return a partial frame, and does not leak a stack trace from inside the model's code.

Layer 2 — restricted execution

Code that passes static analysis is still not trusted. It runs via `exec` against a **copy** of the frame (`df = X.copy()`) in a namespace that exposes only `df`, `pd`, and `np`, with a minimal `_SAFE_BUILTINS` allowlist — arithmetic and aggregation helpers like `abs`, `min`, `max`, `sum`, `round`, `len`, `range`, and the numeric constructors, and nothing that performs I/O. The escape hatches (`__import__`, `open`, `eval`) simply do not exist in scope. The contract is pandas/numpy transforms only — the CAAFE pattern, never arbitrary capability.

The result contract is then enforced in order, each failure a typed `FeatureExecutionError`: a non-`DataFrame` result raises `Feature code must leave a pandas DataFrame in df`; a frame with no new columns raises` Feature code added no new column; a non-numeric new column raises` New feature <name> is not numeric. Surviving new columns have ±inf replaced with NaN` so downstream estimators never receive a malformed frame.

Layer 3 — the tiered sandbox

Layers 1 and 2 run in-process; they block the obvious capabilities, but a determined escape against a CPython process is never something to bet sensitive data on. For untrusted data, escalate isolation with `execution.sandbox`, whose type is exactly `Literal["monty", "docker", "e2b", "local"]`, defaulting to `"monty"`:

`sandbox`	Isolation	Use for
`local`	None — host process, restricted exec only	trusted data, dev only

`sandbox`	Isolation	Use for
`monty`	In-process restricted interpreter (default)	typical CAAFE on owned data
`docker`	OS-level container, no host network/FS	untrusted data
`e2b`	Remote ephemeral microVM	untrusted data at higher assurance

Beyond the strongest sandbox sits **HITL** (human-in-the-loop): when `execution.require_approval` is `True` (the default), generated code is surfaced for human approval before it runs. This is the final tier — a person, not a policy, signs off. The other knob, `execution.timeout_seconds`, defaults to `60`.

> ⓘ **Defaults are the safe end of every axis**
>
> Out of the box, `genai.enabled = False`; when enabled, `sandbox = "monty"`, `timeout_seconds = 60`, `require_approval = True`, and `cost_benefit_gate = True`. You loosen these deliberately, per profile, never globally — and only `local` removes isolation entirely.

> ◆ **Two orthogonal gates: how much, and what**
>
> The `CostBenefitGate` is a *governance* control — it limits spend and runaway agentic loops. The executor and sandbox are *security* controls — they govern what the model's output may do. `cost_benefit_gate` governs **how much** the model runs; the executor and sandbox govern **what** its output may do. Neither substitutes for the other — and the LLM proposes only candidate text, never a capability.

PART III

The hands-on guide

A complete, runnable walkthrough — from uv add and a booted app to a governed GenAI loop and a served model, ending in an end-to-end worked example.

Part III · Practice · Chapter 13

Install & boot your first app

Go from `uv add` to a booted application and a working `firefly-ds` CLI in minutes — installing only the extras you actually use.

Firefly DataScience is a hexagonal, secure-by-default metaframework, and the core stays import-light. Heavy libraries (pandas, scikit-learn, XGBoost, MLflow, …) live behind optional extras and load lazily, so you install what you need and nothing else. The only hard dependency is the Firefly Agentic GenAI substrate; everything else is an extra. **Python 3.13+** is required.

Install an extra

Pick the extra that matches what you are doing. Extras compose — combine them in a single brackets clause, e.g. `uv add "fireflyframework-datascience[tabular,tracking,genai]"`.

```
# Core only — ports, the application bootstrap, the DI container.
uv add fireflyframework-datascience

# Classical tabular — pandas, numpy, scikit-learn, xgboost, lightgbm, catboost,
optuna.
uv add "fireflyframework-datascience[tabular]"

# AutoML stack — the curated bundle: tabular + tabfm + autogluon + tracking +
validation + data.
uv add "fireflyframework-datascience[automl-stack]"

# GenAI — script execution, embeddings, vector stores via Firefly Agentic.
uv add "fireflyframework-datascience[genai]"

# Everything — tabular, DL, NLP, mlops, serving, lineage, orchestration, genai.
uv add "fireflyframework-datascience[full]"
```

Boot the application

`FireflyDataScienceApplication` mirrors the PyFly / Spring Boot lifecycle: load config → print banner → build the DI container → discover and apply auto-configurations → eagerly initialize singletons → return a ready `ApplicationContext`. One call constructs and starts it.

```
from fireflyframework_datascience import FireflyDataScienceApplication

# Construct and start in one call.
app = FireflyDataScienceApplication.run()

print(app.bean_count)                    # number of wired beans, len(app.container)
print(app.config.default_ml_framework)   # the active ML framework -> "sklearn" by
default
print(app.applied_auto_configurations)   # the auto-config classes that matched, in
@order
```

`run(**kwargs)` forwards to the constructor. The common options point it at a config directory, select active profiles, and silence the banner — handy when a test asserts on clean output.

```
app = FireflyDataScienceApplication.run(
    config_dir="./config",        # directory containing firefly-datascience.yaml
    profiles=["local"],           # active configuration profiles
    print_output=False,           # silence the banner + wiring summary
)
```

With `print_output` left on (the default), the application prints a short wiring summary after start — your `profiles`, `beans`, `auto-config` count, `ml framework`, `genai` state, and `sandbox`. The defaults reflect the classical-first stance: `genai` is **off** and the sandbox is `monty`. Resolve wired components from the container by type:

```
from fireflyframework_datascience.models import TrainerPort

trainers = app.container.resolve_all(TrainerPort)   # all registered trainers
config = app.get(type(app.config))                  # or app.config directly
```

The `firefly-ds` CLI

Installing the package exposes the `firefly-ds` command (run it with `uv run firefly-ds <cmd>`). Three subcommands cover the day-one workflow: print the version, check the environment, and introspect a booted app.

```
# Print the framework version.
firefly-ds version

# Check the environment and report which adapter extras are installed.
firefly-ds doctor

# Boot the app and list applied auto-configurations + registered beans.
firefly-ds introspect

# introspect with explicit config and profiles.
firefly-ds introspect --config-dir ./config --profile local --profile gpu
```

`doctor` first verifies that the required Firefly Agentic substrate is present, then prints an *installed / partial / not-installed* status for every optional extra — the fastest way to confirm your environment before a run. `installed` means every representative module for the extra resolves; `partial` means some do; `not installed` means none.

✓ Expected — `firefly-ds doctor`

```
Firefly DataScience doctor — v0.1.0
  python : 3.13.1 (macOS-15.5-arm64-arm-64bit)
  agentic: ok (required)

Optional adapter extras:
  tabular       installed      7/7
  tabfm         not installed  0/1
  tracking      installed      1/1
  validation    installed      1/1
  data          partial        1/2
  genai         installed      2/2

Your rows depend on which extras you installed.
```

▲ If `agentic` is MISSING

The Firefly Agentic substrate is the one hard dependency. If `doctor` reports `agentic: MISSING`, the application will not boot — reinstall the package (the base install pulls Agentic in).

Part III · Practice · Chapter 14

Data, validation & classical AutoML

Wrap your data in a `Dataset`, validate it, then let `AutoML` cross-validate a set of trainers and crown a winner — all classical, all reproducible.

The `Dataset` is the single container the rest of the framework hands around: features `X`, an optional target `y`, the `task`, and metadata. It is import-light — pandas and scikit-learn are imported lazily — so the type is usable before any extra is installed. We start from sklearn's breast-cancer data, a **binary** task, which means `roc_auc` is the default selection metric.

Build a `Dataset`

```
import pandas as pd
from sklearn.datasets import load_breast_cancer

from fireflyframework_datascience.automl import AutoML
from fireflyframework_datascience.core.types import TaskType
from fireflyframework_datascience.datasets import Dataset

raw = load_breast_cancer(as_frame=True)
X: pd.DataFrame = raw.data
y = raw.target

dataset = Dataset(
    name="breast_cancer",
    X=X,
    y=y,
    task=TaskType.BINARY,                # breast cancer is binary -> roc_auc by
default
    target_name="target",
    feature_names=list(X.columns),
)
```

The required fields are only `name` and `X`; everything else has a default (`y=None`, `task=TaskType.CLASSIFICATION`). Setting `task=TaskType.BINARY` explicitly is what selects `roc_auc` as the metric the gate and AutoML maximize throughout the run.

Validate before you train

`BasicValidator` catches empty data, all-null or constant columns, duplicate rows, and null targets *before* you waste time training. It returns a report with an `ok` flag.

```
from fireflyframework_datascience.validation.adapters import BasicValidator

report = BasicValidator().validate(dataset.X, dataset.y)
assert report.ok                          # no all-null columns, no null target, etc.
```

Split and run AutoML

Hold out a test split the usual way. For a classification task with a target present, `train_test_split` stratifies on `y` automatically; both arguments are keyword-only.

```
train, test = dataset.train_test_split(test_size=0.25, random_state=42)

# Cross-validate candidates and fit the winner.
result = AutoML(cv=5, n_trials=20, random_state=42).fit(train)

print(result.best_model.name)   # winning trainer
print(result.best_score)        # winner's CV score
for entry in result.leaderboard:
    print(entry)                # "<model>  <metric>=<score>"

preds = result.predict(test.X)  # predict with the fitted winner
report = result.evaluate(test)  # holdout metrics
```

With no other arguments, `AutoML` uses the default trainers (`RandomForestTrainer`, `LinearTrainer`, `HistGradientBoostingTrainer`, plus the boosting libraries if installed), a default `SklearnMetricsEvaluator`, and a `DefaultSearchPolicy`. Each candidate is wrapped in an impute-and-scale preprocessing pipeline, cross-validated, and the winner refit on the full training set. The leaderboard is sorted best-first; each entry stringifies as the model name followed by `<metric>=<score>`.

✓ Expected — leaderboard

On breast-cancer, a linear model tops a leaderboard with `roc_auc` near 0.99:

```
LinearTrainer           roc_auc=0.9952
RandomForestTrainer     roc_auc=0.9789
HistGradientBoostingTrainer roc_auc=0.9761
```

Exact scores depend on the data, CV splits, and trial budget; the format is fixed.

There is a declarative path too. `AutoML.from_context(app, cv=5, n_trials=20)` pulls its trainers, evaluator, search policy, validator, and tracker straight from the application container, so an app's auto-configured (or custom) adapters are used automatically — each component falling back to its default when not registered.

ⓘ CV score vs. holdout score

The leaderboard prints **cross-validation** scores on the training data, while `evaluate(test)` reports metrics on the untouched holdout. The two measure different things, so they will not match — and that is by design.

Production-grade selection & explainability

The same three lines scale to a production-grade selector: choose an explicit CV strategy, pick the winner on the metric that matches the problem, calibrate its probabilities, stack the top candidates, and explain the result — all with keyword switches that compose.

```
from sklearn.model_selection import StratifiedKFold

result = AutoML(
    cv=StratifiedKFold(4, shuffle=True, random_state=0),  # TimeSeriesSplit /
GroupKFold also work
    calibrate=True,                  # trustworthy probabilities (reports
brier_score)
    ensemble=True, ensemble_size=3,  # stack the top-3 candidates
).fit(train, metric="average_precision")            # select on PR-AUC, not accuracy

print(result.best_model.name)                       # "stacking_ensemble"
print(result.evaluate(test).metrics["brier_score"]) # calibration quality

# Explain the winner — deterministic global importances (SHAP if the extra is
installed).
for name, importance in result.explain(test).top(8):
    print(f"{name:<26} {importance:+.4f}")
```

✓ Run it end-to-end

The `samples/advanced_automl.py` script runs exactly this on the real breast-cancer data — a calibrated stacking ensemble selected on PR-AUC, global explainability, and a persisted audit trail of every feature decision. It runs offline, and uses a real LLM to propose features when a key is present. A test covers it, so it is guaranteed to work.

Part III · Practice · Chapter 15

GenAI feature engineering, gated

A language model proposes feature code; classical cross-validation decides what survives. The LLM never touches the score — the data does.

Firefly treats generative AI as a *proposer*, never a judge. `GenAIFeatureEngineer` runs a strict four-step loop — **propose → execute → measure → gate** — and only folds a feature into the working frame if a `CostBenefitGate` confirms it measurably lifts the cross-validated score. Because everything is injectable, the loop runs fully offline with a deterministic stand-in.

Propose features as code

A `FeatureProposer` returns `FeatureProposal`s — a `name`, a line of pandas that adds a column to a DataFrame `df`, and a `rationale`. Here a `StaticFeatureProposer` stands in for the LLM with a fixed, reproducible set: one genuine signal and one feature that should be thrown away.

```
from fireflyframework_datascience.features import FeatureProposal,
StaticFeatureProposer
from fireflyframework_datascience.features.genai import GenAIFeatureEngineer

proposer = StaticFeatureProposer([
    FeatureProposal(
        "debt_to_income",
        "df['debt_to_income'] = df['loan_amount'] / (df['income'] + 1)",
        "DTI",
    ),
    FeatureProposal("noise", "df['noise'] = 0.0", "should be rejected"),
])

engineered = GenAIFeatureEngineer(proposer, cv=4).engineer(train)
print(engineered.summary())
```

For each proposal the engineer **executes** the code safely against a copy of the frame, **measures** its cross-validation lift with an estimator, and **gates** the result. `debt_to_income` reconstructs a real signal and lifts the score, so it is accepted; the constant `noise` column adds nothing, so the gate rejects it. Each accepted feature is folded into the working frame, so the next proposal is measured against the *improved* baseline.

✓ Expected

GenAI feature engineering: 1 accepted, 1 rejected; roc_auc 0.7875 -> 0.7889 (lift +0.0013)

`engineered.accepted` lists `debt_to_income`; `engineered.rejected` lists `noise` with the reason `no lift (0.7889 <= 0.7889)`. The lift is small but positive and real — the gate rejects anything that does not strictly beat the running baseline.

◆ **The cost-benefit gate decides**

`CostBenefitGate.accepts(current, candidate)` returns `True` only when `candidate - current > min_gain`. With the default `min_gain=0.0`, any strict improvement is kept; raise it to demand features that clear a meaningful bar. The LLM proposes — the measured score decides, and a proposal that does not beat the seeded classical baseline is rejected.

Reading the result

`engineer()` returns an `EngineeringResult` with the engineered dataset plus a full audit trail: `baseline_score`, `final_score`, `lift`, and the `metric`. Each `AcceptedFeature` carries its `proposal`, `score`, and the `gain` over the previous best; each `RejectedFeature` carries the `proposal`, a `reason`, and the candidate `score`.

```
for acc in engineered.accepted:   # AcceptedFeature
    print(acc.proposal.name, acc.score, acc.gain)

for rej in engineered.rejected:   # RejectedFeature
    print(rej.proposal.name, rej.reason, rej.score)
```

▲ **Proposed code never runs raw**

`FeatureCodeExecutor` is not `eval`-on-faith. A static safety analysis first denies imports, dunder access, and dangerous builtins (`eval`, `exec`, `open`, `__import__`, …); the vetted snippet then runs in a restricted namespace exposing only `df`, `pd`, and `np`. Unsafe, failing, or non-numeric output becomes a `RejectedFeature` rather than crashing the loop.

The `StaticFeatureProposer` above is the offline stand-in for the LLM. With a real model you swap in `AgentFeatureProposer`, which wraps a `FireflyAgent` and is built lazily — no LLM client is created at startup. The loop is identical; only the source of proposals changes. Turning on the real model is covered later in this part, under *Serving & turning on a real LLM*.

A persisted audit trail

The in-memory `EngineeringResult` carries the trail for one run. For governance you usually want it **persisted** — so a risk or compliance function can reconstruct *why* a feature was kept or dropped, long after the run. Wire an `AuditLogPort` and every gate decision (accepted *or* rejected, with the score, baseline, and reason) is appended durably.

```
from fireflyframework_datascience.features.audit import JsonlAuditLog

engineer = GenAIFeatureEngineer(proposer, audit_log=JsonlAuditLog("audit/genai-
decisions.jsonl"))
engineer.engineer(train)
# audit/genai-decisions.jsonl now holds one JSON line per decision — greppable and
append-only.
```

◆ Every GenAI decision is logged and auditable

This is what makes that claim literally true. `JsonlAuditLog` writes one JSON object per decision — `feature`, `code`, `decision`, `score`, `baseline`, `metric`, and a human reason — so the record of what the model proposed and why it was kept or dropped outlives the process. `AuditLogPort` is a port: swap the JSONL adapter for your own sink (a database, an event bus) without touching the loop.

Part III · Practice · Chapter 16

The agentic loop in practice

An LLM proposes a solution; the classical engine trains and cross-validates it; a deterministic verifier — not "it ran" — decides whether it earned its place.

The agentic loop realizes the same propose-and-gate pattern at the level of whole models. Its cycle is **propose → train/CV → verify → reflect → select**. A proposer yields a `SolutionCandidate` (a trainer plus hyperparameters); `AgenticAutoML` cross-validates it; and a `DeterministicVerifier` rules on whether the result genuinely beats a trivial baseline before it can be selected.

Run a fixed search plan

For tests and reproducible runs, `SequenceProposer` replays a fixed candidate list — the first is the seed, the rest are dispensed one per reflection round. Here we seed the three default trainers.

```
from fireflyframework_datascience.engineering import SequenceProposer,
SolutionCandidate
from fireflyframework_datascience.engineering.loop import AgenticAutoML

proposer = SequenceProposer([
    SolutionCandidate("linear"),
    SolutionCandidate("random_forest"),
    SolutionCandidate("hist_gradient_boosting"),
])

run = AgenticAutoML(proposer, cv=3, max_iterations=4).solve(train)
print(run.summary())
```

Each candidate is trained, cross-validated with sklearn's `cross_val_score`, and **verified** by a `DeterministicVerifier` — it must clear a trivial `DummyClassifier(strategy="prior")` baseline by a margin, not merely run. A failing candidate scores `-inf` and never aborts the loop. Only a valid verdict that improves on the running best can take the lead.

✓ Expected

Agentic AutoML: 3 attempts (3 verified); best=linear roc_auc=0.7897 (baseline 0.5000)

All three seeded candidates clear the `roc_auc=0.5000` trivial baseline, so all three are verified; `linear` wins.

◆ Correctness ≠ ran

A candidate that trained and returned a score has only proven it *executed*. Verification is a separate stage: the `DeterministicVerifier` demands a finite score that beats the trivial baseline by a `margin`. Anything else is rejected — execution-success is never mistaken for correctness.

The audited trail

`solve` returns an `EngineeringRun` — a full, auditable trace plus the refit best model. `run.attempts` is every iteration; `run.valid_attempts` are the ones that passed verification; `run.summary()` renders the one-line recap. The loop is greedy with two budgets: `max_iterations` (default 8) caps reflection rounds, and `patience` (default 3) stops the search once it stalls.

```
run.best_candidate    # SolutionCandidate | None
run.best_score        # float (nan if nothing verified)
run.model             # the refit Model (None if nothing verified)
run.baseline_score    # the trivial baseline it had to beat
run.valid_attempts    # only the verified attempts

for a in run.attempts:
    print(a.candidate.trainer, a.score, a.verdict.valid, a.verdict.reason)
```

> ⓘ **Which trainers are allowed**
>
> The `trainers` list a proposer sees comes from the loop's registry: by default `linear`, `random_forest`, and `hist_gradient_boosting`, plus `xgboost`, `lightgbm`, and `catboost` when those optional libraries are installed. Pass `trainers=...` to `AgenticAutoML` to constrain or extend the search space.

Part III · Practice · Chapter 17

Serving & turning on a real LLM

Serve the winner in-process with zero dependencies — then flip three environment variables to let a real LLM do the proposing, with the gate and verifier still in charge.

A fitted `Model` is served by a `ModelServerPort`. The zero-dependency default, `LocalModelServer`, loads the model in the host process and answers `predict` / `predict_proba` — nothing leaves the process, so a model that scored well in development behaves identically when served.

Serve the model

```
from fireflyframework_datascience.serving import LocalModelServer

server = LocalModelServer()           # server.name == "local"
server.load(result.best_model)
prediction = server.predict(test.X.iloc[[0]])   # score one applicant
print(int(prediction[0]))
```

Calling `predict` before `load` raises `FireflyDataScienceError("No model loaded — call load(model) first")`. Heavier servers — for example `BentoMLModelServer` behind the `serving` extra — implement the *same* `name`/`load`/`predict` surface, so swapping the backend is a one-line change, never a rewrite.

Turn on a real LLM

Everything so far ran offline with deterministic stand-ins. GenAI is **off by default** — `GenAIConfig.enabled` is `False`, and a fresh install is fully deterministic. To let a real model do the proposing, set your key, enable GenAI, and choose a model string. Config keys map to env vars with the prefix `FIREFLY_DATASCIENCE_` and the nested delimiter `__`.

```
export OPENAI_API_KEY=sk-...                      # or ANTHROPIC_API_KEY=sk-ant-...
export FIREFLY_DATASCIENCE_GENAI__ENABLED=true
export FIREFLY_DATASCIENCE_GENAI__DEFAULT_MODEL=openai:gpt-4o   # or
anthropic:claude-sonnet-4-5
```

Then swap the deterministic stand-ins for the agent-backed proposers. Use `AgentFeatureProposer` in place of the `StaticFeatureProposer` from the GenAI feature-engineering step, and `AgentSolutionProposer` in place of the `SequenceProposer` from the agentic-loop step. Nothing else changes — both wrap a `FireflyAgent` built lazily on first use, so the application still boots without a key.

```
from fireflyframework_datascience.features.genai import AgentFeatureProposer,
GenAIFeatureEngineer
from fireflyframework_datascience.engineering.loop import AgenticAutoML,
AgentSolutionProposer

# Feature engineering, now LLM-proposed.
engineered = GenAIFeatureEngineer(AgentFeatureProposer(model="openai:gpt-4o"),
cv=4).engineer(train)

# The agentic loop, now LLM-reflected.
run = AgenticAutoML(AgentSolutionProposer(model="anthropic:claude-sonnet-4-
5")).solve(train)
```

◆ The LLM proposes; the classical engine decides

Switching on a real model changes only the *source* of proposals. The `CostBenefitGate` still accepts a feature only when its cross-validation score beats the seeded baseline by `min_gain`, and the `DeterministicVerifier` still rules on every candidate. The model's confidence is irrelevant — the measured score is the sole arbiter.

▲ Cost gate + secure execution still apply

Set `genai.budget_usd` to enforce a hard spend ceiling for a run (default: no ceiling), and keep API keys in a git-ignored `.env` or a secrets manager — the framework never logs them. LLM-proposed feature code still runs only through `FeatureCodeExecutor`'s static safety analysis and restricted namespace; choose the sandbox tier under `execution.sandbox` (default `monty`).

Part III · Practice · Chapter 18

End-to-end: the Lumen credit-risk build

One focused story: a lending model whose default risk is secretly driven by a feature the model is never handed — and the framework rediscovers it, gates it, and serves the winner.

The `samples/lumen_credit_risk.py` sample ties every preceding chapter together on a realistic synthetic lending dataset, with **no LLM API key required**. Default is *latently* driven by **debt-to-income** (`loan_amount / income`) — the ratio shapes the labels but is deliberately withheld from the model. The pipeline has three acts: GenAI feature engineering, classical AutoML, and in-process serving.

The build, step by step

19. **Synthesize the data.** `make_lending_dataset` builds a `Dataset` of six raw columns (`income`, `loan_amount`, `age`, `employment_years`, `credit_history_length`, `num_prior_defaults`). DTI carries the largest coefficient in the logit (`4.0`), yet never appears as a column. The task is `TaskType.BINARY`, so the default metric is `roc_auc`.
20. **Propose features as code.** A `StaticFeatureProposer` emits `debt_to_income`, `loan_per_year_employed`, and a constant `noise` feature. In production an `AgentFeatureProposer` asks an LLM; the stub exercises the *exact same* gate offline.
21. **Engineer and gate.** `GenAIFeatureEngineer` runs propose → execute → measure → gate. `debt_to_income` reconstructs the latent driver and lifts the score, so the `CostBenefitGate` keeps it; `noise` adds nothing and is rejected.
22. **Select with AutoML.** `AutoML(cv=4).fit(engineered.dataset)` cross-validates the default trainers and crowns a winner.
23. **Replay accepted code on the holdout.** The accepted feature code is re-executed on the test frame so train and test schemas stay identical, then `result.evaluate(...)` reports holdout metrics.
24. **Serve and score.** `LocalModelServer` loads the winner in-process and scores a live applicant.

Acts 1 and 2 — discover DTI, reject noise, pick a winner

```
from fireflyframework_datascience.automl import AutoML
from fireflyframework_datascience.core.types import TaskType
from fireflyframework_datascience.datasets import Dataset
from fireflyframework_datascience.features import FeatureProposal,
StaticFeatureProposer
from fireflyframework_datascience.features.genai import GenAIFeatureEngineer


def _logistic_scorer(task: TaskType):
    from sklearn.linear_model import LogisticRegression
    return LogisticRegression(max_iter=1000)


dataset = make_lending_dataset()
train, test = dataset.train_test_split(test_size=0.25, random_state=0)

proposer = StaticFeatureProposer([
    FeatureProposal("debt_to_income", "df['debt_to_income'] = df['loan_amount'] /
(df['income'] + 1)", "DTI"),
    FeatureProposal("loan_per_year_employed",
                    "df['loan_per_year_employed'] = df['loan_amount'] /
(df['employment_years'] + 1)", "burden"),
    FeatureProposal("noise", "df['noise'] = 0.0", "should be rejected"),
])

# A fast LogisticRegression measures lift; cv=4 sets the folds.
fe = GenAIFeatureEngineer(proposer, scorer_estimator=_logistic_scorer, cv=4)
engineered = fe.engineer(train)

automl = AutoML(cv=4)
result = automl.fit(engineered.dataset)
print(result.best_model.name)         # the winning trainer
print(result.leaderboard_table())
```

In production the proposer is `AgentFeatureProposer` — the real LLM-backed class — not a stub. The story is identical: the LLM proposes, the gate measures, and only earned features survive.

Acts 3 — replay, evaluate, and serve

The accepted features were measured on the train fold, so the *same code* is re-executed on the test frame via `FeatureCodeExecutor`. Re-applying the code (not the values) is what keeps train and test schemas identical — `engineered.accepted` is the audit trail that makes the replay exact. The winner is then served in-process to score a new applicant.

```
from fireflyframework_datascience.features.executor import FeatureCodeExecutor
from fireflyframework_datascience.serving import LocalModelServer

executor = FeatureCodeExecutor()
working = test.X.copy()
for accepted in engineered.accepted:
    working = executor.execute(accepted.proposal.code, working)

engineered_test = test.with_features(working)
evaluation = result.evaluate(engineered_test)
print(evaluation.metrics)            # holdout metrics dict

server = LocalModelServer()
server.load(result.best_model)
applicant = engineered_test.X.iloc[[0]]
print(int(server.predict(applicant)[0]))   # 0 = no default, 1 = default
```

✓ Expected

=== Lumen Lending — credit-risk AutoML ===
accepted features : ['debt_to_income', 'loan_per_year_employed']
rejected features : ['noise']
feature-eng lift : +0.0XYZ
winning model : <trainer name>
holdout metrics : {'accuracy': ..., 'roc_auc': ...}
applicant predicted default = 0

Exact numbers vary with your scikit-learn version, but the shape is stable: `debt_to_income` is accepted, `noise` is rejected, the lift is positive, and a winner is served.

◆ The governance thesis, end to end

This is the whole pattern in one run: the framework rediscovers the signal you deliberately hid, the cost-benefit gate throws away the feature that adds nothing, and a deterministic engine — never the LLM — selects what reaches production. GenAI accelerates; measured improvement governs.

IV

PART IV

Reference

The evidence base, the configuration and CLI surface, the broader Firefly ecosystem, an honest read on maturity, and a glossary.

Part IV · Reference · Chapter 19

Benchmarks & evidence

Every published number is produced by a bundled, runnable harness — fixed seed, default trainers, no manual tuning. Measured, reproducible, honestly reported.

A framework that fuses GenAI with classical ML earns trust only by measurement. Firefly DataScience separates *how it proves it is good* from *how it loads data day-to-day*: the same `DatasetLoaderPort` that powers a quick `iris` smoke test in CI also pulls real OpenML suites for credibility runs. This chapter describes the evaluation strategy and reproduces the real results. Every figure here comes from a script in `benchmarks/` — the full table lives in `benchmarks/RESULTS.md`.

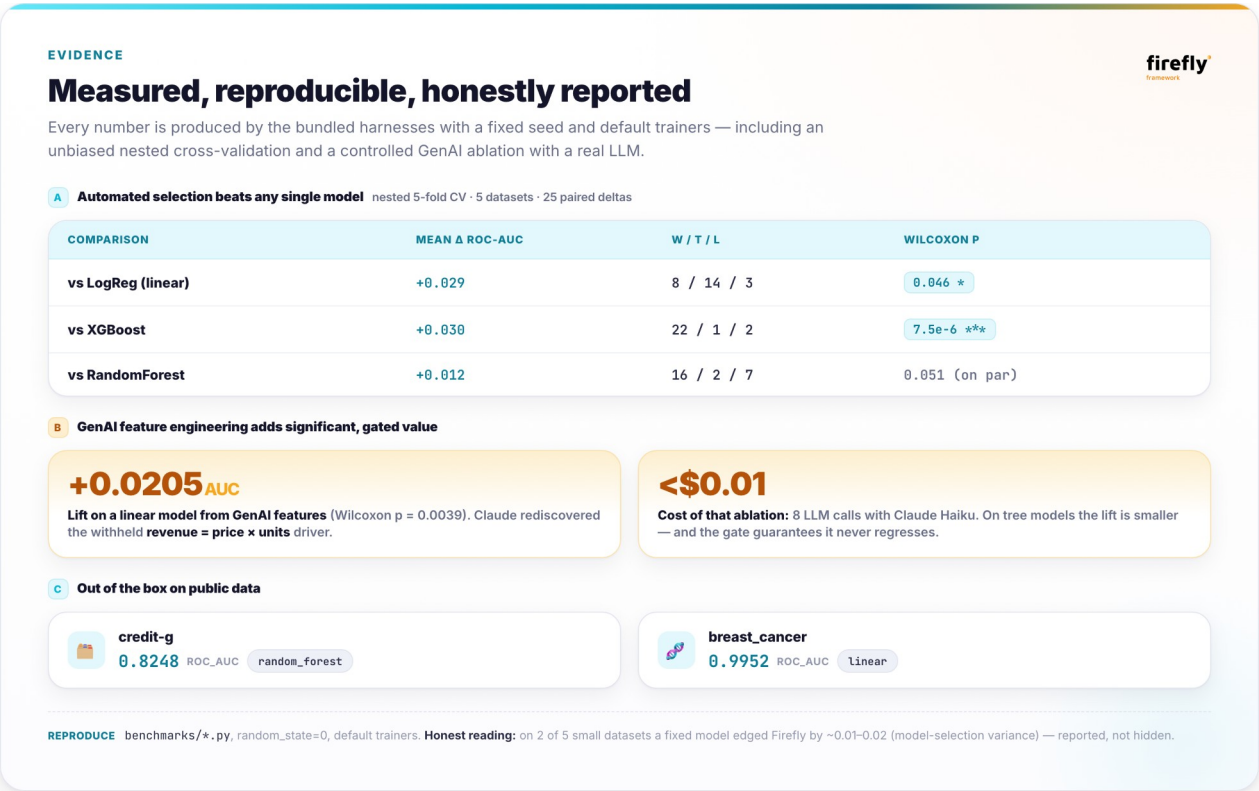


Figure 11. *Measured, reproducible, honestly reported.*

A three-tier evaluation strategy

Proving the framework and running it day-to-day are different jobs, so the benchmarks are split into three tiers — credible public suites, fast CI smoke datasets, and agentic capability suites. Only Tier 2 runs without network access, which is why it backs the default CI gate; Tiers 1 and 3 define how the framework earns external credibility over time.

Tier	Purpose	Sources	When it runs
Tier 1 — Credibility	Compare against the literature on standard suites	AMLB, OpenML-CC18, OpenML-CTR23	Offline / scheduled (network)

Reproducing the runs

There is nothing to take on faith. Install the extras and run the harnesses; the Tier-2 suite is offline and finishes in seconds, while the Tier-1 suite fetches real OpenML tasks over the network.

```
uv sync --extra tabular --extra data --extra validation

# Tier-2 (offline, no network) — scikit-learn built-ins
uv run python benchmarks/automl_benchmark.py

# Tier-1 (OpenML, needs network) — AMLB-style on real categorical data
uv run python benchmarks/amlb_benchmark.py
```

Tier-2 — offline suite

CI-smoke datasets shipped with scikit-learn. `AutoML(cv=3)` cross-validates the default trainers (RandomForest, Linear, HistGradientBoosting; plus XGBoost / LightGBM / CatBoost when installed), fits the winner, and reports a holdout score — all with a fixed `random_state=0`.

Dataset	Task	Metric	CV	Holdout	Winner
breast_cancer	binary	roc_auc	0.9939	0.9952	linear
iris	multiclass	accuracy	0.9467	1.0000	random_forest
wine	multiclass	accuracy	0.9700	1.0000	linear
diabetes	regression	rmse	−54.10	56.46	linear
california_housing	regression	rmse	−0.473	0.455	hist_gradient_boosting

Tier-1 — OpenML-CC18 (AMLB-style)

Real OpenML tasks with genuine categorical data (e.g. `credit-g`), exercising the dtype-aware preprocessing and string-target encoding. `AutoML(cv=5)`, holdout ROC-AUC. The numbers are comparable to published AutoGluon / H2O / FLAML results on the same datasets — out of the box, on real data, with no manual tuning.

OpenML id	Dataset	Metric	CV	Holdout	Winner
31	credit-g	roc_auc	0.7689	0.8248	random_forest
37	diabetes	roc_auc	0.8155	0.8724	linear
1464	blood-transfusion	roc_auc	0.7465	0.7511	linear

OpenML id	Dataset	Metric	CV	Holdout	Winner
1480	ilpd	roc_auc	0.7347	0.7798	linear

The full AMLB (104 tasks), CC18 (72) and CTR23 (35) suites plug into the same `run_amlb` shape under a nightly compute budget — each suite is just a curated set of OpenML ids, so a credibility run is "load each id, fit, score, compare".

Beating the baseline — and the bias in that claim

`benchmarks/beat_baseline.py` pits Firefly's AutoML against a default `LogisticRegression` — the common single-model reference — on 5-fold cross-validated ROC-AUC. Firefly **wins or ties on 6/6** datasets: it never does worse than the baseline, because it selects the best model from a portfolio that *includes* the baseline's family. It wins clearly where the data is non-linear (`phoneme` **+0.149**) and correctly *matches* the baseline by choosing `linear` where a linear model is genuinely best. Mean gain across the six: **+0.029 ROC-AUC**.

> ▲ **That score is mildly optimistic**
> The 6/6 comparison reports Firefly's *cross-validated selection* score — a maximum over models scored on the same folds, which is optimistically biased. The defensible, **unbiased** version uses nested cross-validation. We report both, and lead with the harder one.

Scientific evaluation — nested cross-validation

`benchmarks/scientific_eval.py` removes the selection bias with **nested 5-fold CV**: an inner CV selects the model on each outer fold's *training* data only, and the untouched outer fold gives the unbiased estimate. Firefly AutoML is compared against three fixed single models on identical folds, with a one-sided Wilcoxon signed-rank test over all 25 paired deltas (5 folds × 5 datasets).

Firefly AutoML vs...	mean Δ ROC-AUC	wins / ties / losses	Wilcoxon p
LogReg (linear)	+0.029	8 / 14 / 3	0.046
RandomForest	+0.012	16 / 2 / 7	0.051 (on par)
XGBoost	+0.030	22 / 1 / 2	7.5e-6

Firefly **significantly beats** a single LogReg ($p = 0.046$) and a single XGBoost ($p \approx 7.5e\text{-}6$), and is **statistically on par with** RandomForest ($p \approx 0.051$) — because it *adapts* per dataset, picking boosting/bagging on the non-linear `phoneme` and `linear` where linear genuinely wins (`blood-transfusion`, `ilpd`).

> ⓘ **The honest reading**
>
> On 2 of 5 small datasets a fixed model edged Firefly out by ~0.01–0.02 — model-selection variance on ~1000-row data. We report this rather than hide it. The headline claim is the defensible one: *automated selection matches or beats any fixed single model, decisively on non-linear data, and never collapses to a poor choice.*

GenAI value — a controlled ablation with a real LLM

`benchmarks/genai_value.py` isolates the contribution of GenAI feature engineering. The dataset is a retail "high-value customer" task whose true driver is **revenue = unit_price × units** — a multiplicative interaction withheld from the model that a *linear* learner cannot derive on its own. Four systems, 8 repeated train/test splits, the real `anthropic:claude-haiku-4-5`.

System	ROC-AUC (mean ± std)
linear (raw)	0.9752 ± 0.006
linear + GenAI	0.9957 ± 0.002
Firefly AutoML (raw)	0.9929 ± 0.003
Firefly AutoML + GenAI	0.9950 ± 0.003

- **GenAI lift on a linear model: +0.0205 ROC-AUC** (0.9752 → 0.9957) — **Wilcoxon *p* = 0.0039**, significant. Claude proposed and the gate accepted `total_revenue` / `price_volume_ratio`, rediscovering the withheld multiplicative driver from the schema alone.
- On Firefly's tree-based AutoML the lift is smaller (**+0.002**): trees already approximate the interaction, so there is less for GenAI to add — and the cost/benefit gate guarantees it never regresses.
- **Cost:** 8 LLM calls, well under **\$0.01** with Claude Haiku.

> ◆ **The LLM proposes; the classical engine decides**
>
> GenAI proposes feature code; a deterministic classical engine measures the cross-validated lift; and a cost/benefit gate keeps only what is proven on the data. That is why the ablation can only improve or stay neutral — never regress. The benchmarks measure both the classical core and the gated accelerator on the same footing.

> ✓ **Reproducible by construction**
>
> Every number in this chapter is produced by a bundled benchmark harness with a fixed `random_state=0` and default trainers — no manual tuning. Runs are seed-pinned, decisions are logged, and lineage is captured end-to-end, so results can be reproduced and audited, not just believed.

Part IV · Reference · Chapter 20

Configuration & the CLI

One typed settings model resolves your whole app — constructor args, environment, `.env`, and layered YAML, in that order.

`FireflyDataScienceConfig` is a `pydantic-settings` model. Load it once at startup with `FireflyDataScienceConfig.load(...)` and everything — ML framework, tracking, GenAI, code execution, banner — resolves from a single, predictable precedence chain. The model is classical-first by design: GenAI is **off** until you turn it on, and even then every GenAI call sits behind a cost/benefit gate.

```
from fireflyframework_datascience.core.config import FireflyDataScienceConfig

config = FireflyDataScienceConfig.load(config_dir="config", profiles=["prod"])
print(config.default_ml_framework)  # "sklearn"
print(config.genai.enabled)         # False
```

Resolution precedence

Values resolve from highest priority to lowest. Anything set at a higher level wins; a missing source is simply skipped. This ordering comes straight from `settings_customise_sources`, which returns `(init_settings, env_settings, dotenv_settings, *reversed(yaml_sources), file_secret_settings)` — earlier sources win, and reversing the YAML list lets profile overlays outrank the base file.

Priority	Source	How to set it
1 (highest)	Constructor kwargs	values passed to `FireflyDataScienceConfig(...)`
2	Environment variables	prefixed `FIREFLY_DATASCIENCE_`, nested via `__`
3	`.env` file	same naming as environment variables
4	Profile YAML overlays	`firefly-datascience-<profile>.yaml` (later profiles outrank earlier)
5	Base YAML	`firefly-datascience.yaml`
6 (lowest)	Field defaults	the defaults shown below

Setting values: environment and YAML

Every field is reachable from the environment using the `FIREFLY_DATASCIENCE_` prefix; nested models use the `__` delimiter once per level. Constructor kwargs beat env, `.env`, YAML, and defaults — useful in tests.

```
# Environment beats both YAML files and the field default:
export FIREFLY_DATASCIENCE_DEFAULT_ML_FRAMEWORK=pytorch
export FIREFLY_DATASCIENCE_GENAI__ENABLED=true        # nested via __
export FIREFLY_DATASCIENCE_EXECUTION__SANDBOX=docker

# Constructor kwargs beat everything (tests):
config = FireflyDataScienceConfig(default_ml_framework="xgboost")
assert config.default_ml_framework == "xgboost"
```

A profile is just a named YAML overlay. Keep a base file with shared settings, then add one overlay per environment or hardware target and activate them by name; later profiles outrank earlier ones for any key both set.

```
# config/firefly-datascience.yaml  (base — lowest YAML priority)
app_name: lumen-ds
default_ml_framework: sklearn
tracking_enabled: false
banner:
  mode: TEXT
genai:
  enabled: false
  default_model: openai:gpt-4o
  cost_benefit_gate: true
execution:
  sandbox: monty
  timeout_seconds: 60
  require_approval: true

# config/firefly-datascience-prod.yaml  (overlay for profile "prod")
tracking_enabled: true
genai:
  enabled: true
  budget_usd: 25.0
```

Configuration fields

The top-level model carries the framework defaults and three nested sub-models — `banner`, `genai`, and `execution`. The defaults below are the lowest-priority source: every one can be overridden by YAML, `.env`, an environment variable, or a constructor kwarg.

Field	Type	Default
`app_name`	`str`	`"firefly-datascience-app"`

Field	Type	Default
`profiles`	`list[str]`	`[]` (populated by `load`)
`default_ml_framework`	`str`	`"sklearn"`
`default_dataset_backend`	`str`	`"pandas"`
`tracking_enabled`	`bool`	`False`
`model_registry_url`	`str | None`	`None`
`feature_store_endpoint`	`str | None`	`None`

The nested sub-models encode the governance posture. GenAI is **off by default** and gated; generated code runs in a sandbox (`monty`) with a timeout and an approval gate.

Sub-model field	Type	Default
`genai.enabled`	`bool`	`False`
`genai.default_model`	`str`	`"openai:gpt-4o"`
`genai.cost_benefit_gate`	`bool`	`True`
`genai.budget_usd`	`float | None`	`None`
`execution.sandbox`	`Literal["monty","docker","e2b","local"]`	`"monty"`
`execution.timeout_seconds`	`int`	`60`
`execution.require_approval`	`bool`	`True`
`banner.mode`	`BannerMode`	`TEXT`

The `firefly-ds` CLI

Installing the package exposes the `firefly-ds` command (run with `uv run firefly-ds <cmd>`). Three subcommands cover the whole loop: confirm the version, check the environment, and inspect a booted application.

```
# Print the framework version.
firefly-ds version

# Check the environment and report which adapter extras are installed.
firefly-ds doctor

# Boot the app and list applied auto-configurations + registered beans.
firefly-ds introspect

# introspect with explicit config and profiles.
firefly-ds introspect --config-dir ./config --profile local --profile gpu
```

Command	What it does	Flags
`version`	Print the framework version.	—
`doctor`	Verify the Agentic substrate, then report installed / partial / not-installed status for every optional extra.	—
`introspect`	Boot the app; list applied auto-configurations and registered beans.	`--config-dir`, `--profile` (repeatable)

✓ doctor is the fastest pre-flight check

`doctor` verifies that the required Firefly Agentic substrate is present, then prints an installed / partial / not-installed status for every optional extra (`tabular`, `tabfm`, `automl`, `dl`, `nlp`, `tracking`, `validation`, `featurestore`, `serving`, `lineage`, `orchestration`, `data`, `genai`) — the fastest way to confirm your environment before a run. If it reports `agentic: MISSING`, the application will not boot; reinstall the package.

Part IV · Reference · Chapter 21

The Firefly ecosystem & where it stands today

Firefly DataScience is one member of a coherent, open framework family — so an investment here compounds across the whole ecosystem, not a single tool.

Firefly DataScience does not stand alone. It is built on the broader **Firefly Framework** and its agentic runtime, and it shares the DNA — auto-configuration, dependency injection, a hexagonal core, common CI gates — that every member of the family carries. That is what lets an investment in one member pay off across the rest.

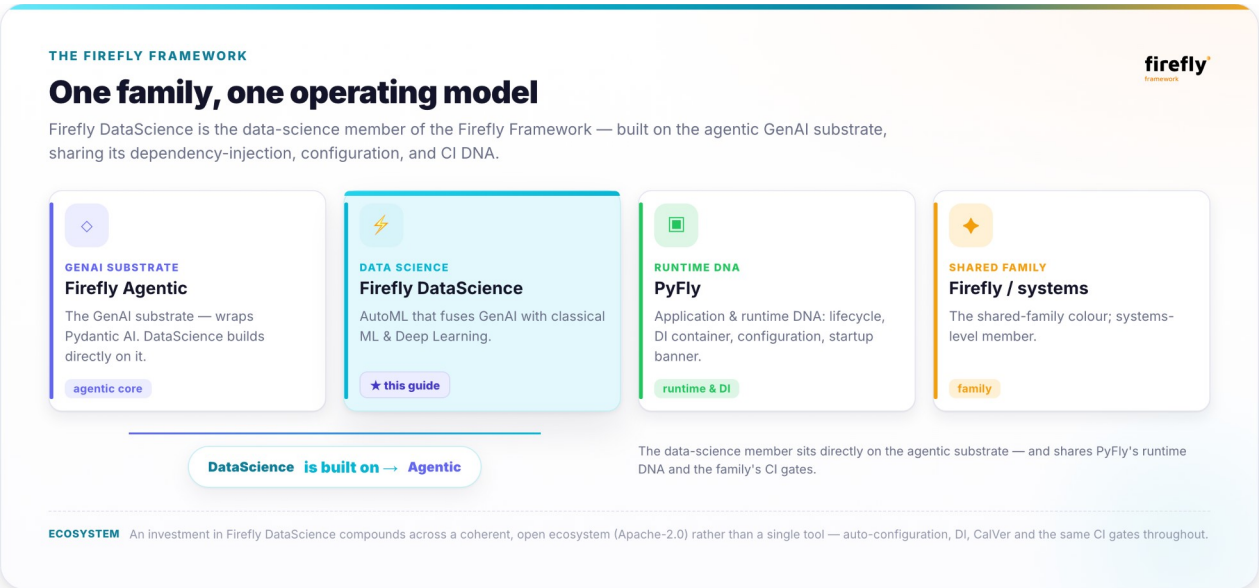


Figure 12. *Firefly DataScience within the Firefly Framework family.*

The family

Each member owns a layer of the stack and exposes the same Firefly-native conventions, so they compose cleanly rather than colliding.

- **Firefly Agentic** — the GenAI substrate. It wraps Pydantic AI and is the one hard dependency of DataScience; the gated GenAI accelerator builds directly on it.
- **PyFly** — the application and runtime DNA: the Spring-Boot-style lifecycle, the DI container, and the layered configuration model that DataScience mirrors.
- **The systems-level member** — the lower-level infrastructure layer the family sits on.
- **Firefly DataScience** — the AutoML metaframework described in this guide, fusing GenAI with classical ML and Deep Learning behind hexagonal ports.

Firefly-native by construction

Because the members share a common foundation, the same guarantees hold across the family: a lean DI container wires ports to adapters, auto-configuration discovers them from packaging entry points, and the

domain never depends on a vendor SDK. The shared conventions are not cosmetic — they are what make an adapter swappable and a config portable between members.

- **Auto-configuration** — adapters self-register under an entry-point group and are wired conditionally on the optional extra being installed.
- **Dependency injection** — resolution by type annotation, with constructor injection and fail-fast eager initialization.
- **CalVer** — calendar-based versioning across the family.
- **Shared CI gates** — the same green-build discipline applies to every member.
- **Apache-2.0** — open source, self-hostable, no vendor or library lock-in.

> ◆ **The thesis carries across the family**
>
> The architecture is what makes the rule enforceable. GenAI lives in **adapters** behind ports; the deterministic classical engine trains, scores and selects. A GenAI adapter can only ever *propose* — the container resolves a port, and the classical engine decides whether a proposal survives a measured improvement over a seeded baseline.

> ⓘ **Where it stands today**
>
> Firefly DataScience is a working, openly-developed framework (Apache-2.0) with a green continuous-integration pipeline and results validated on public benchmarks. It is young — best suited to teams that want to own and shape their AI stack, rather than a mature, off-the-shelf product with a catalogue of customer case studies. That is precisely where a strategic early adopter captures the most value.

APPENDIX

Glossary & list of figures

A short reference for the recurring terms and a numbered index of every figure in this guide.

The definitions below are deliberately compact — enough to anchor a term as it appears in the preceding chapters, faithful to how the framework actually uses it.

Glossary

Term	Meaning
AutoML	Automated machine learning: cross-validating a portfolio of trainers, fitting the winner, and ranking every candidate in a leaderboard — here, classical and deterministic.
Cost-benefit gate	The check that keeps a GenAI proposal only if it yields a measured cross-validated improvement over the seeded baseline; on by default.
Hexagonal architecture	Ports-and-adapters design: the domain depends on protocol interfaces, never on a vendor SDK, so implementations are swappable.
Port / adapter	A *port* is a `Protocol` the domain calls; an *adapter* is a concrete class that implements it. The container binds them by type annotation.
Dependency injection	Wiring components by type annotation rather than by hand — constructor injection, with circular-dependency detection and fail-fast boot.
Auto-configuration	A `@configuration` class discovered via packaging entry points that contributes adapters conditionally on an optional extra being importable.
CAAFE	Context-Aware Automated Feature Engineering: an LLM proposes feature code from the dataset schema; the gate keeps only the features that measurably help.
Agentic loop	Propose → train → score → select, driven by the agentic runtime, with a measured improvement required at every step.
Sandbox / monty	The isolated runtime that LLM-generated code executes in, with a timeout and an approval gate; `monty` is the secure default (also docker, e2b, local).

Term	Meaning
Leaderboard	The best-first ranking of every candidate model from an AutoML run, each entry stringified as `<model> <metric>=<score>`.
TaskType	The light core enum naming the learning task — `BINARY`, `MULTICLASS`, `REGRESSION` — used to pick the default metric.
Modality	The light core enum naming the data modality (tabular, text, vision, timeseries, multimodal) a model and its adapters operate on.
Lineage	The end-to-end record of data, decisions and runs — what was proposed, the score it earned, and the keep-or-discard verdict — captured so results can be audited.
Cross-validation	Estimating model performance by repeated train/validate splits; *nested* CV selects the model on inner folds and scores it on an untouched outer fold for an unbiased estimate. `cv` accepts any scikit-learn splitter — e.g. `TimeSeriesSplit` (temporal) or `GroupKFold` (grouped) — to prevent leakage.
Probability calibration	Post-processing a classifier so its predicted probabilities match observed frequencies (`calibrate=True`, cross-fit `CalibratedClassifierCV`); quality is read off the **Brier score** (lower is better).
Stacking ensemble	Combining the top-k leaderboard candidates behind a cross-fit meta-learner (`ensemble=True`) — the standard last-mile lift over single-best selection.
PR-AUC (average precision)	Area under the precision–recall curve; the right selection target for imbalanced binary problems, via `metric="average_precision"`, where ROC-AUC over-credits the majority class.
Explainability	Attributing a model's predictions to its features with deterministic, peer-reviewed methods — permutation importance (default) or SHAP (local + global) — never an LLM. Exposed as `result.explain(dataset)`.
Audit trail	A durable, append-only record of every GenAI gate decision (kept or dropped, with score and reason) via an `AuditLogPort` such as `JsonlAuditLog`.

List of figures