ROBIN

Reusable **O**pen **B**ioacoustic **IN**frastructure

A Vision and Design Overview

ROBIN provides a common, open source foundation for running bioacoustic models, comparing their performance, and using expert-provided annotations to create better models. It is entirely storage and compute agnostic.

What this document is

A short summary of why ROBIN exists, what it does and should do, and how it's designed. It is for project managers, faculty advisors, and potential partners. Appendix 3 contains more detailed technical design information. It is written as if the whole system exists, but it is still in development. The final section describes the current state of the implementation.

Contents

1	Why ROBIN	2
2	Design overview	3
3	Adapter stacks	5
4	Run lifecycle	6
5	Looking forward	7
A1	SoundHub integration	8
A2	Governance & provenance	11
A3	Technical design	14

1 What does ROBIN do, and why do we need it?

A shared foundation lets monitoring programs spend more time on science and less on rebuilding pipelines.

Bioacoustic monitoring programs collect tens or hundreds of terabytes of audio, far more than people can listen to. To find species in that much audio, programs use foundation models like BirdNET or Perch, which automatically find species calls. But it is hard to run these models at scale, keep track of thousands of runs, compare their outputs, and fine-tune models to perform better in the local ecoregions monitoring programs care about. Separate researchers, labs, and institutions each build software for this *ad hoc*, around both particular models and the platforms they use for storage and compute.

To best do bioacoustic monitoring, programs need to do more than just run existing models. Foundation models such as BirdNET and Perch are trained from recordings collected around the world. A monitoring program usually cares about a smaller set of local species, recorded at its own sites with its own equipment, and global foundation models can miss these species or confuse them with others. Monitoring programs can develop better models through an iterative cycle: run models on their catalog of recordings, quantify how reliable model predictions are for each species of interest, use this information to prioritize which clips to validate, and use these expert-validated audio clips to improve models for the taxa they are trying to monitor and

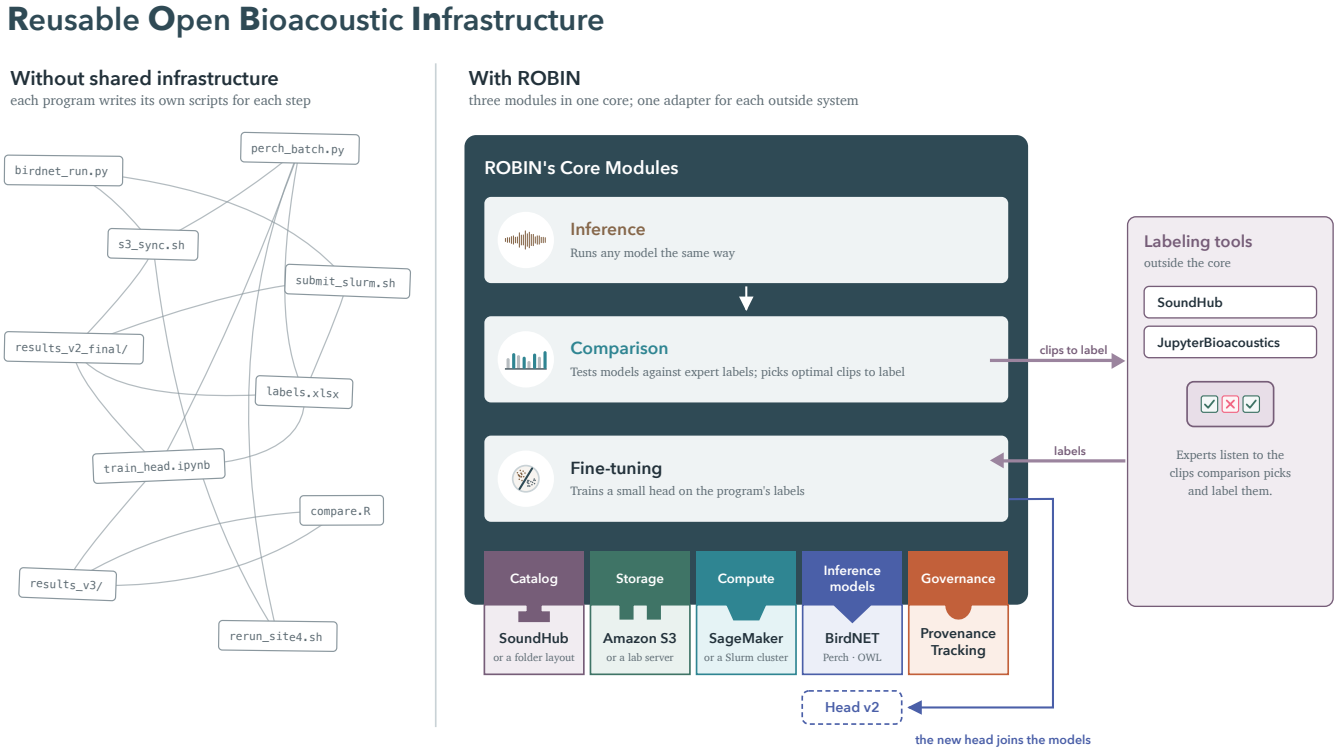


Figure 1 Without shared infrastructure, every program rebuilds the same pipeline. ROBIN provides one core that every program can reuse.

evaluate the updated model on separate labeled recordings to check whether performance improves.

Without shared infrastructure, every step in this cycle is a separate script. A program trying to implement this cycle today manually maintains code for each step: one script to run BirdNET, another for Perch, another to copy audio from S3, another to send jobs to a cluster, Excel files with manual labels, a notebook for fine-tuning, and folders of results with names like `results_v2_final_final`. These pieces depend on each other in ways nobody wrote down, and adding a new model, a new storage system, or running on a different cluster breaks them.

Building and maintaining this code distracts ecologists from doing the science that large-scale bioacoustic data has the potential to enable.

ROBIN exists to solve this problem, by providing a common, open-source foundation for running bioacoustic inference at scale, validating and comparing model performance, and fine-tuning models for specific taxa — all entirely agnostic to where data is stored and what compute platform it is running on.

2 Design overview

One reusable core connects to each program's tools through a small set of adapters.

ROBIN has a core consisting of several parts, each of which are separate Python modules:

1. **Shared contracts** define the files and data formats that every part of ROBIN uses: what a model must report about itself, what a run request contains, and what a result contains.
2. **The run manager** keeps the record of all work: what was requested, on which recordings, with which settings, and what came out. The record makes every result traceable and prevents duplicate work.
3. **The inference engine** runs any model in the same way. It reads audio, splits it into windows, runs the model, checks the output, and writes scores, detections and embeddings in a standardized format. A new model gets all of this work for free, so adding a model means writing one small adapter, not a new pipeline.
4. **The comparison module** measures how well models perform against expert labels, and compares models against one another. It also finds the clips that should be a priority for experts to label next to best improve models using *active learning* criteria.
5. **The fine-tuning engine** trains small classifiers on top of a backbone model (BirdNET, Perch) using a program's own labels, with the aim of improving performance on local species of interest.

ROBIN separates its core from every outside system (e.g. S3 storage, a Slurm cluster). The core talks to the external world through five fixed interfaces, called **ports**, for storage, compute, recording catalogs, inference models, and governance. For each outside system, an **adapter** translates communication with the outside system into the data types ROBIN's core expects. There are five ports that surround ROBIN's core:

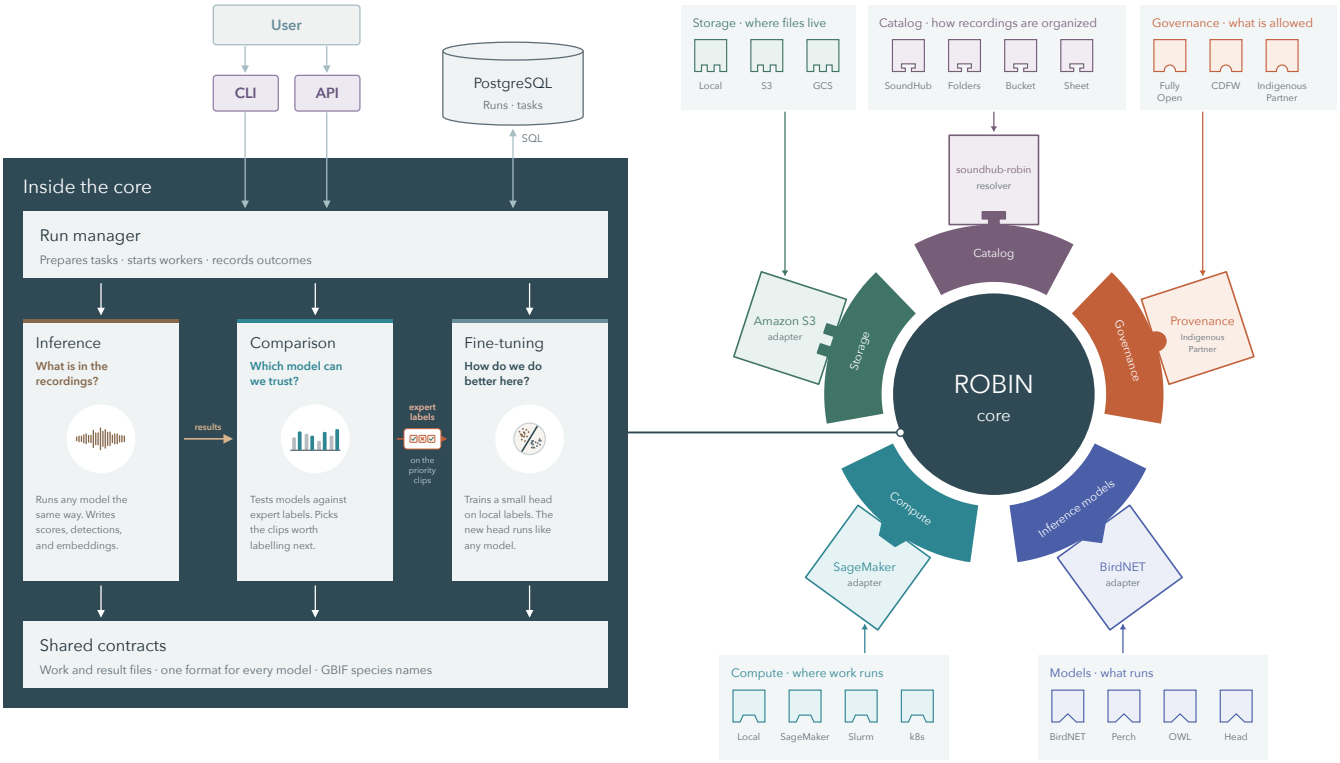


Figure 2 ROBIN's core modules sit inside five ports. Adapters translate between each port and an outside system.

1. **Compute** launches jobs, stops them, and checks their status on a local machine, AWS SageMaker, a Slurm cluster, or a Kubernetes cluster.
2. **Storage** reads and writes audio, work files, and results from the local disk, S3, or other cloud platforms.
3. **Catalog** resolves the URIs of the recordings to process: for example, from a folder layout, an S3 bucket prefix, or the SoundHub API.
4. **Inference models** connect the core inference engine to the model-specific code for e.g. BirdNET, Perch, the PNW Spotted Owl model, or custom classifier heads.
5. **Governance** connects ROBIN to a set of rules for governance, data usage, and provenance tracking. This enables ROBIN to check whether a proposed use is permitted, and track what specific recordings are being used for downstream. This will be the least covered in this document; the governance appendix explains the motivation in more detail.

This design means support for a new environment is one adapter, not a complete rewrite. And because the core is the same everywhere, a result means the same thing on a laptop as on AWS.

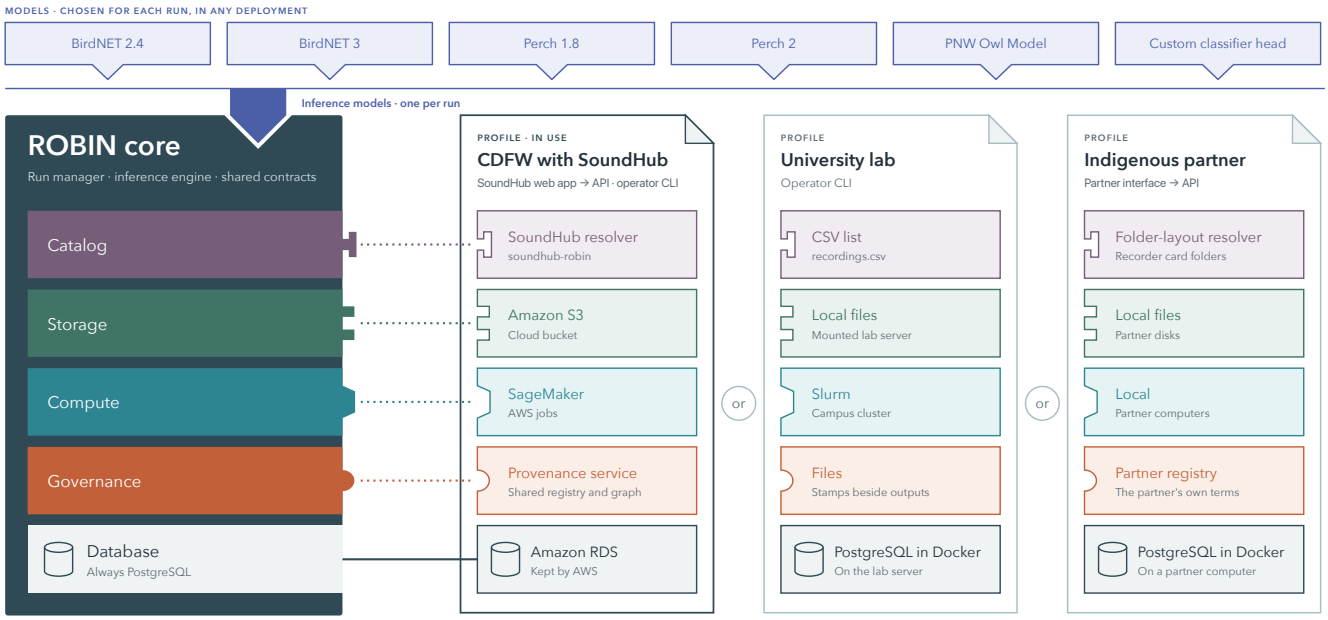


Figure 3 Three deployments, one core. Each profile picks an adapter for every port.

3 One core, different adapter stacks for different users

The same core fits different infrastructure, budgets, and data-governance needs.

Shared infrastructure is only useful if it can effectively be used by monitoring programs that are very different from each other. A state agency, a university lab, and an Indigenous community have different budgets, different computers, and different rules about who may hold their data. If each of them needs its own version of the software, we're back to separate, *ad hoc* systems. Each deployment fits ROBIN to its setting with a **deployment profile**: a YAML file that picks one adapter for each port and names the host of the run manager database.

Users create their ROBIN deployments in a repo, called e.g. `soundhub-robin`, where they define their deployment profile in YAML. ROBIN ships adapters for most common use cases, so most programs set up a deployment by writing YAML, not code.

Different users and programs will store and organize their recordings in different ways. Most catalogs fit a built-in adapter, like defined nested folder layouts or an S3 prefix. However, a larger system like SoundHub may need its own small adapter. In this case, `soundhub-robin` would also include an adapter, so CLI requests query the SoundHub API for all the recordings from e.g. a specific project, or all the recordings for a specific deployment within a certain year.

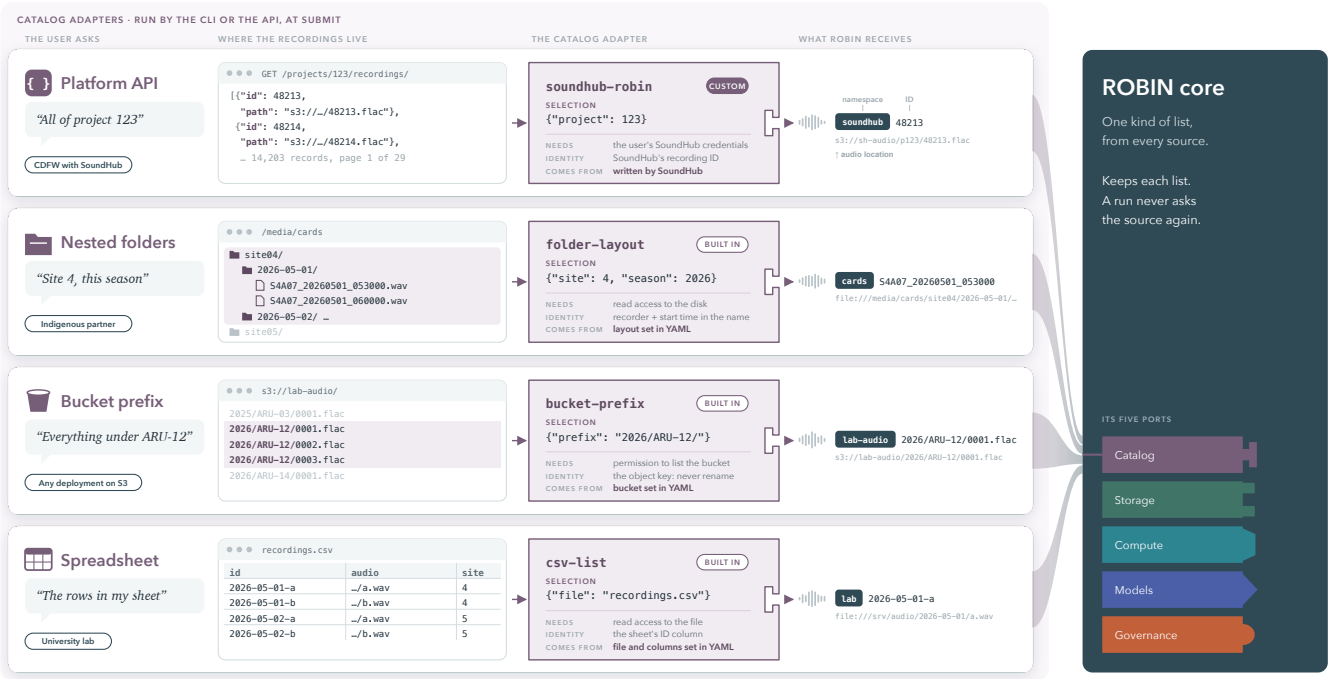


Figure 4 Catalog adapters turn a user's selection into an explicit list of recordings.

4 Run lifecycle

Every request becomes traceable work, from the selected recordings to the final results.

At the scale of tens or hundreds of terabytes, one request becomes thousands of jobs that run for hours or longer. Some jobs fail. Some recordings are damaged. People rerun work and change settings. Without a careful record, a program loses track of what has run, pays to process the same audio again, and cannot say which result is the most recent one. ROBIN's run lifecycle is designed so it's always known what ran, what failed, and which result is current.

A **run** is one request to apply one model to a set of recordings. A user submits the request through a web app, the CLI, or the API. A catalog adapter turns the user's selection into a recording list, and the run manager freezes that list and divides eligible recordings into **tasks**. For each task, it writes a work file to storage, which says which model to run, with which settings, on which recordings. The compute adapter then launches one worker for each task. Each worker reads its work file, runs the inference engine, and writes its outputs and a results file. The run manager reads each result file, checks it, and records the outcome in the database (Figure 5).

Three rules make this reliable at scale:

1. **Only the run manager changes the record.** Workers never connect to the database; they only exchange files with the run manager.
2. **Each recording gets its own outcome.** One damaged recording does not fail the whole task. The run manager tracks which recordings succeeded and which failed, and why.

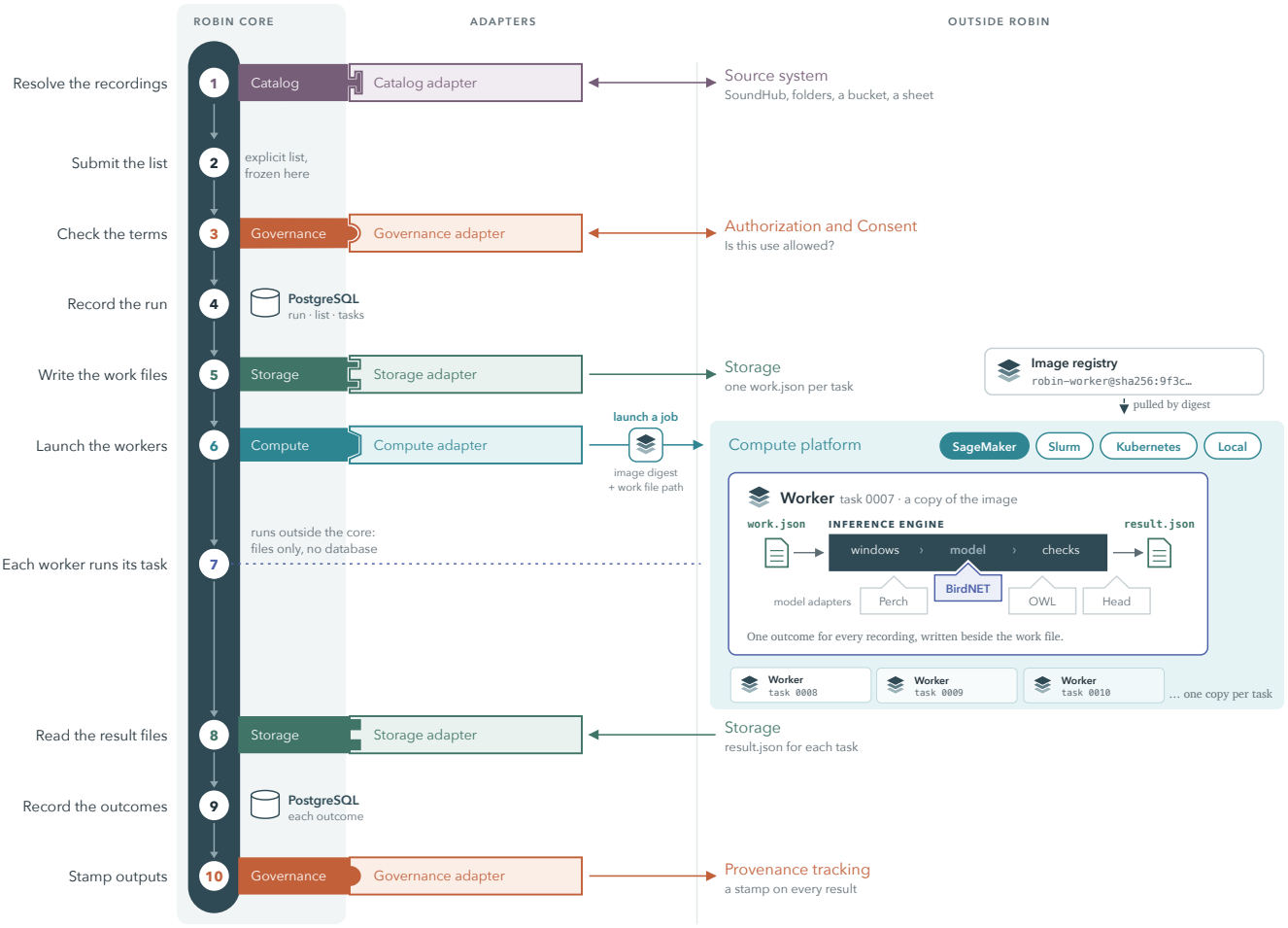


Figure 5 How a run moves through ROBIN, from resolving recordings to stamping outputs.

3. **Avoiding duplicate, unnecessary runs.** ROBIN keeps at most one valid result for each recording and unique work request (model adapter + selected custom settings). Users can directly ask questions like “Which models ran on this recording, and with which settings?” and “Which result is the current one?”

5 Looking forward

Inference and run management are developed; comparison and fine-tuning are next.

ROBIN is still in development. The core inference module, run manager, and the adapters for the storage and compute used by the legacy `soundhub_model_runner` (AWS S3 for storage, SageMaker for compute) are developed. The development of the CLI is ongoing. The first major milestone will be using ROBIN for a run of Perch on the entire historical catalog of CDFW in SoundHub. After that, development on the comparison and fine-tuning modules will be the next step.

Appendix 1

Integrating ROBIN into SoundHub

This appendix describes how ROBIN works with SoundHub, and the ways the two can be joined to work together better later.

The present

In the present, ROBIN operates entirely independently, outside of a thin wrapper repository `soundhub-robin` that defines a deployment configuration and the custom adapter for the catalog port that resolves recording URIs using the SoundHub API. Inference runs are then submitted via the ROBIN CLI, which talks to a very lightweight AWS Aurora RDS instance as its database. When ROBIN finishes an inference run, a path to the S3 URI of the resulting detection Parquet is stored in the database. The `soundhub_api_demo` implementation that uses `api_dock` can then query that database to get the path to the Parquet file to return when SoundHub requests the detections for a particular model and recording.

ROBIN never replaces a file it has written, so every run keeps its own outputs. To give readers one place to look, the controller also copies the current detections for each configuration into a single dataset, partitioned by recording. A read for a whole project then reads one hive-partitioned dataset. The database still decides which result is current; the dataset is a copy of that decision.

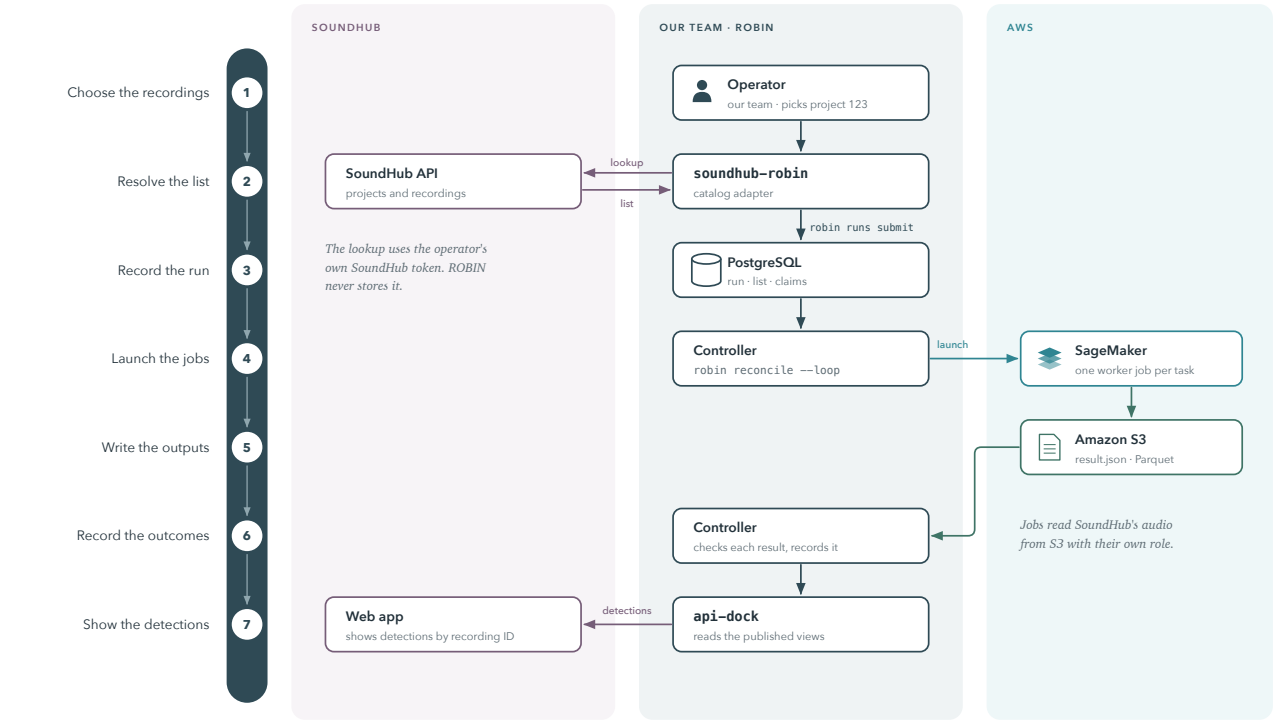


Figure 6 Today: ROBIN runs beside SoundHub, and `api_dock` reads results from ROBIN's database.

Later

Option 1. A user clicks, for example, "Run BirdNET on this project". SoundHub's backend checks the user's access, builds the explicit list of URIs, and sends `POST /v1/runs` with the work request as the JSON body to ROBIN's API server. SoundHub's backend is a client of that server; it does not host ROBIN code.

This is, in effect, a small microservice architecture. In practice, the likeliest way to make this smooth is to extend the current way `api-dock` is used to act as a proxy that handles parsing the request from SoundHub and constructing the proper JSON body to send to ROBIN's API.

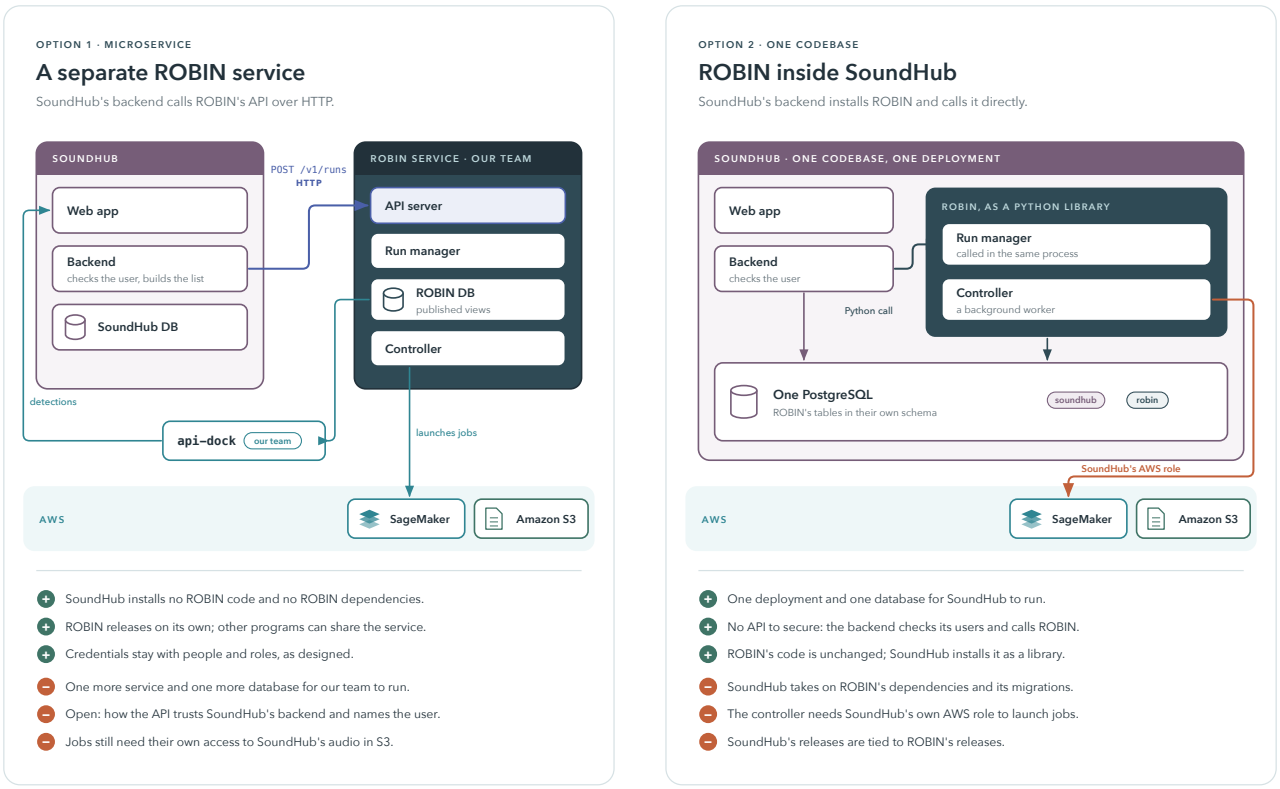


Figure 7 Later: SoundHub submits runs to ROBIN's API through `api-dock`.

Option 2. SoundHub could integrate the run management system and database into its own PostgreSQL database. This saves a separate DB deployment. Further, it could also integrate the API into its own server, so the deployment of ROBIN that serves SoundHub doesn't run as a separate service.

For an application this size, a monolithic application may be preferable and cheaper to host, but there is a cost/benefit analysis here as to how much refactoring the SoundHub backend would require to support this.

Appendix 2

Governance and provenance tracking

A major goal of DSE's work is to build tools that enable Indigenous Data Sovereignty (IDSov). As part of that work, the Scoping Plan Draft pitches a system for data governance and provenance tracking that follows CARE and OCAP principles, and enables FAIR principles within the limits they set.

This system has two parts:

1. **A governance model** that translates a data governance agreement into rules that software can check: who can use which data, for what purpose, and what rules apply to the results.
2. **A provenance standard** for tracking how biodiversity data products are produced, so a result can be traced back to its sources, and a source can trace forward everything it contributed to producing. It extends W3C's PROV standard.

This system is a separate project; ROBIN connects to it through its governance port. This appendix explains what this system does, why it is separate from ROBIN, and how ROBIN connects to it.

What the system does

Governance

A governance agreement states what a community permits its data to be used for. The scoping plan divides it into axes:

- **Usage consent.** Which uses are permitted, and for what purpose.
- **Access tiers.** Who can see the data and results derived from it.
- **Ownership and attribution.** Who owns the data and its downstream products, and who must be credited for its use.
- **Location sensitivity.** Whether the place a recording was made can be shown, and at what precision.

Provenance tracking

Provenance answers two questions:

← Backward
What produced this result? Which records, model, settings, under what consent terms, and when.

Forward →
What did this recording help produce? For a recording or a whole deployment, which detections, fine-tuned models, and downstream analyses used it?

Vocabulary

This standard is an extension of W3C PROV, which has three fundamental classes:

Entity	Activity	Agent
A thing.	Makes or changes entities.	The person or system responsible for an activity.

Why the system is separate from ROBIN

1. **It is not specific to bioacoustics.** Most of the standard applies equally to camera traps, eDNA, and museum collections. It should be its own published standard with its own implementations, and it should know nothing about ROBIN or audio.
2. **ROBIN must work without it.** Many ROBIN users will not need governance. ROBIN's own record stays the source of truth for what ROBIN did, and ROBIN never needs the standard to do its work correctly. The consent rules stay out of ROBIN's core.
3. **Different groups need different rules.** One deployment can follow CDFW's rules on sharing recording locations; another can follow a community's consent agreement. A custom adapter for each is simpler than one large system with many settings.
4. **One place for the rules across many tools.** SoundHub, labelling tools, exports, and ROBIN can all ask the same registry and report to the same graph. Each rule is written once, and every governed action is in one audit log.

The last point also shows where each rule is enforced. ROBIN checks a requested use before it starts work. A rule about who may read results, such as hiding recording locations, is enforced by the tool that serves the results, such as `api-dock`. ROBIN's part is to put the terms that apply on every stamp, so the reader can enforce them.

How ROBIN connects

The governance port has two operations:

- **Report.** ROBIN sends a provenance record for each thing it does: a run accepted, a task finished, an output written. Records go out through an outbox that the controller delivers, so a slow or unavailable registry never blocks a run.
- **Check.** Before ROBIN accepts a run, it asks whether the proposed use is permitted. A refusal stops the run before any work starts.

This likely will look like two separate interfaces:

1. **Provenance out**: a sink (ROBIN -> standard). When ROBIN does something, it needs to report what it did. The same database transaction that records an outcome could also write the outgoing provenance records to a provenance_outbox table.
2. **Governance in**: a gate (standard -> ROBIN). ROBIN asks whether a use is permitted, and obeys the answer. Unlike the sink, this **can block** work, so it runs at a clear point and fails clearly.

At this point, designing this interface is not the goal, and will come later after the bulk of ROBIN is built.

In the first SoundHub deployment there is no governance adapter at all (`governance: none`): the IAM permissions that SoundHub sets for each operator and for the job role decide what can be read, and ROBIN keeps only its own record.

Appendix 3

Technical design

A. Packages

ROBIN consists of a small ecosystem of Python packages.

PACKAGE	JOB	MAY IMPORT	TODAY
`robin_contracts`	File formats, port interfaces, shared value types, canonical JSON and digests, and checks that more than one package needs	nothing	Built
`robin_inference_engine`	Runs one model over a list of recordings and writes outputs	contracts	Built
`robin_models`	Model adapters (BirdNET, Perch, OWL, and an ONNX runtime for classifier heads) and the output-version table	contracts	Built
`robin_adapters`	Storage adapters (local disk and S3, through ondio); later the compute adapters	contracts	Storage built, Compute in progress
`robin_worker`	The `robin-worker` command: one work file in, one result file out	contracts, engine, adapters, models	Built
`robin_run_manager`	The record, its SQL, the operations, the profile loader, and the inference configuration	contracts; adapters only when a profile is built with `compose`	In progress
`robin_cli`	The `robin` command	run manager	In progress
`robin_api`	The web API	run manager	Later
`robin_comparison`, `robin_fine_tuning`	Comparing models against expert labels; training heads	not designed	Later

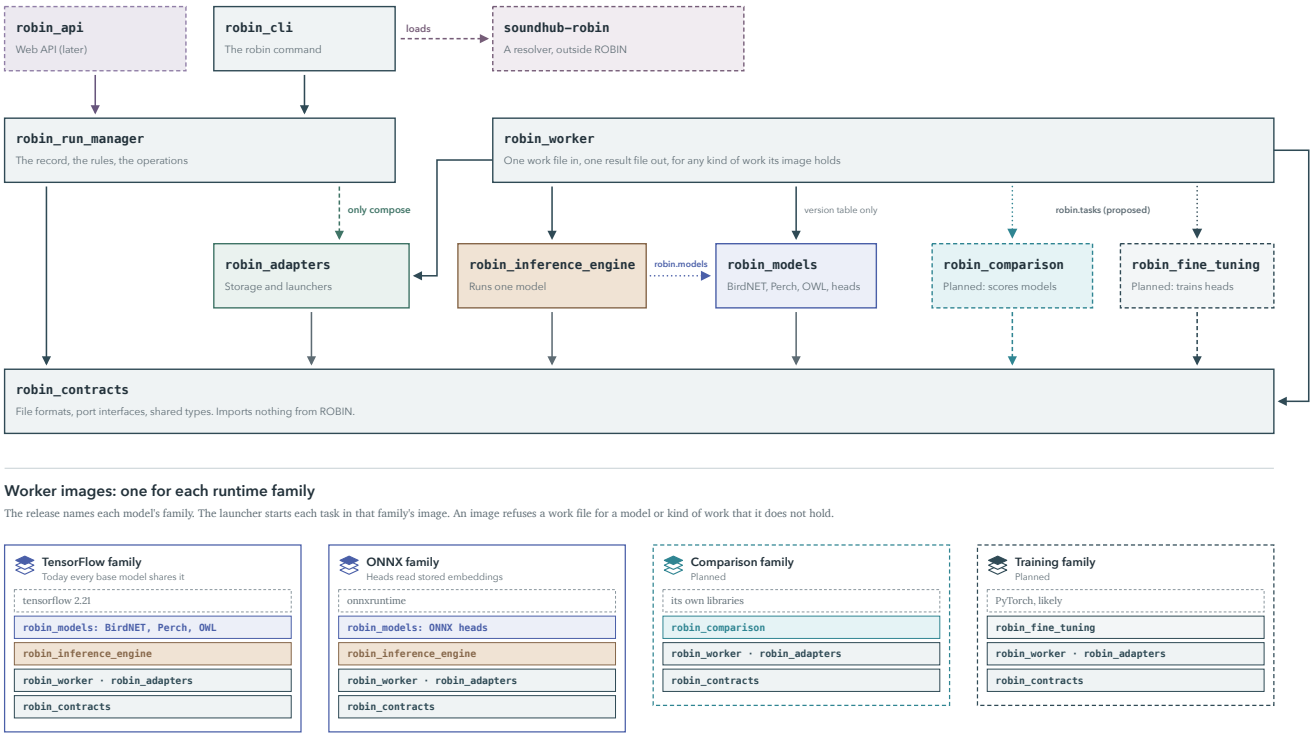


Figure 8 The ROBIN package ecosystem.

B. Contracts and files

The different components of ROBIN exchange files (not function calls), wherever they run on different computers. Different components communicate using a shared set of versioned contracts defined in `robin_contracts`, a package consisting only of Pydantic data classes. These contracts are composed into two primary files which workers use to run inference and describe their results.

The work file (`work.json`) is everything a worker needs for one task: an ID, where to write outputs, the version of the code the worker must have, and a description of the work itself. The work names the model, its files with digests, the recordings, the settings, and the outputs that are requested.

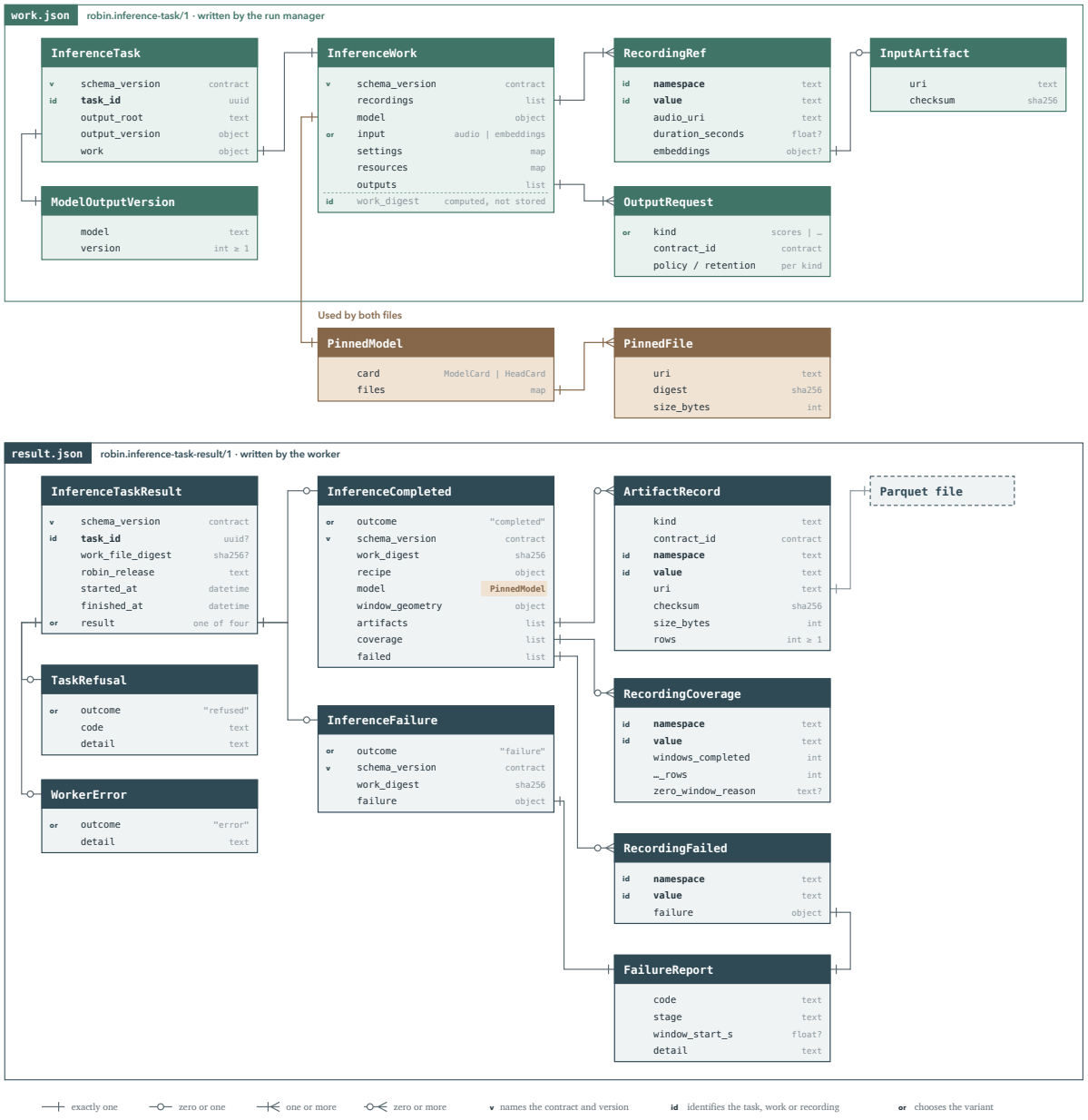


Figure 9 The contracts: two JSON files for each task, each with a named, versioned contract.

The result file (`result.json`) says how the task ended:

- **Completed.** With an outcome for each recording, covered or failed.
- **Failure.** A problem that affected the entire work, e.g. a corrupted model weights file.
- **Refused.** The worker would not start due to a misspecification in the work file.
- **Worker error.** An umbrella around all other unexpected failures.

Outputs. There are three types of outputs, each of which are Parquet files:

- **Embeddings.** The output prior to the last layer from a foundation model (Perch, BirdNET) for a particular window.

- **Scores.** The set of scores for a particular window, associating each taxon with the output score from the model.
- **Detections.** A set of detections in a window derived from scores, by a threshold or by keeping the top *k* labels.

Both scores and detections for each taxon are associated with the GBIF taxon through each model's taxa registry.

A request says how many scores to keep: all of them (`full`), those above a floor (`thresholded`), or the highest *k* in each window (`top_k`). The choice matters at scale. Full BirdNET scores are about 7.8 million rows for each hour of audio (1,200 windows × 6,522 labels), so most runs keep fewer.

Failures. The engine names the stage at which something failed, and each stage has one scope:

- **Whole work.** Validating the request, fetching the model files, loading the taxa registry, and building the model. These run before the first recording, and a failure here ends the task with a *failure* result.
- **One recording.** Fetching the audio, reading an input embeddings file, running the model, checking a window, and writing an output. A failure here is recorded against that recording, and the worker continues with the next one.

A recording too short for even one window is not a failure: it is covered, with zero windows and a reason.

Storage layout. Each run has its own run folder, associated with a UUID, and each task that belongs to its run has its own subfolder within it:

```
<root>/runs/<run>/tasks/<task>/work.json
<root>/runs/<run>/tasks/<task>/result.json
<root>/runs/<run>/tasks/<task>/outputs/...
```

C. Models, cards, and heads

ROBIN runs two kinds of model. A **backbone**, such as BirdNET or Perch, reads audio and produces embeddings, scores, or both. A **head** is a small classifier that reads a backbone's saved embeddings and produces scores. Fine-tuning will produce heads. (A model such as the PNW Spotted Owl model goes straight from audio to scores and has no embeddings.)

Model cards. Each model has a card: a short YAML file that says what the model is. It gives the window length, the sample rate, how audio is downmixed, resampled and padded (in the case of a final window shorter than the full window size), the width of its embeddings, what kind of scores it emits, and the digest of the taxa registry that labels its scores. A card is identified by the hash of its contents. A head's card names the exact backbone card whose embeddings it reads, and ROBIN refuses to pair a head with any other backbone.

Pinned files. The work file (`work.json`) pins each of the model's files by location, SHA-256 digest and size. The engine checks every digest and size before it builds the model. ROBIN never rehosts model weights. They come from either public repositories (Kaggle, HuggingFace, Zenodo) or a user's S3.

Taxa registries. A registry gives each output position of a model its meaning: a label, and where possible a scientific name, a common name and a GBIF taxon key. A label with no GBIF match is published as `unresolved` (15 of Perch v1.8′s 10,932 labels) and is never silently dropped.

Heads. A head is three files: a head card, an ONNX graph and a taxa registry. It runs on the CPU with onnxruntime, so scoring stored embeddings needs no TensorFlow. The graph must include any normalisation it was trained with. ROBIN cannot detect a head that leaves it out; it runs, and gives plausible wrong scores.

Output versions. Each model has an integer *output version*. It goes up in any change that changes that model's outputs, whatever the cause: changes in the adapter, the engine, preprocessing, bug-fix, or a library such as TensorFlow. A change that leaves the outputs the same keeps the version.

Images. Models whose libraries can share an environment form a *runtime family*, and each family has its own worker image. PyTorch for fine-tuning, or a model pinned to an older TensorFlow, gets its own family instead of breaking everyone else's.

D. Profiles, deployments, and credentials

A deployment profile is a YAML file which describes the settings for one deployment of ROBIN: where the database lives, and which adapters to use for each port.

Profiles also refer to the environment variables needed for storage and compute adapters to access their systems. **ROBIN has no access of its own;** it uses the credentials of the person who ran the command. ROBIN doesn't store any of these credentials, ever.

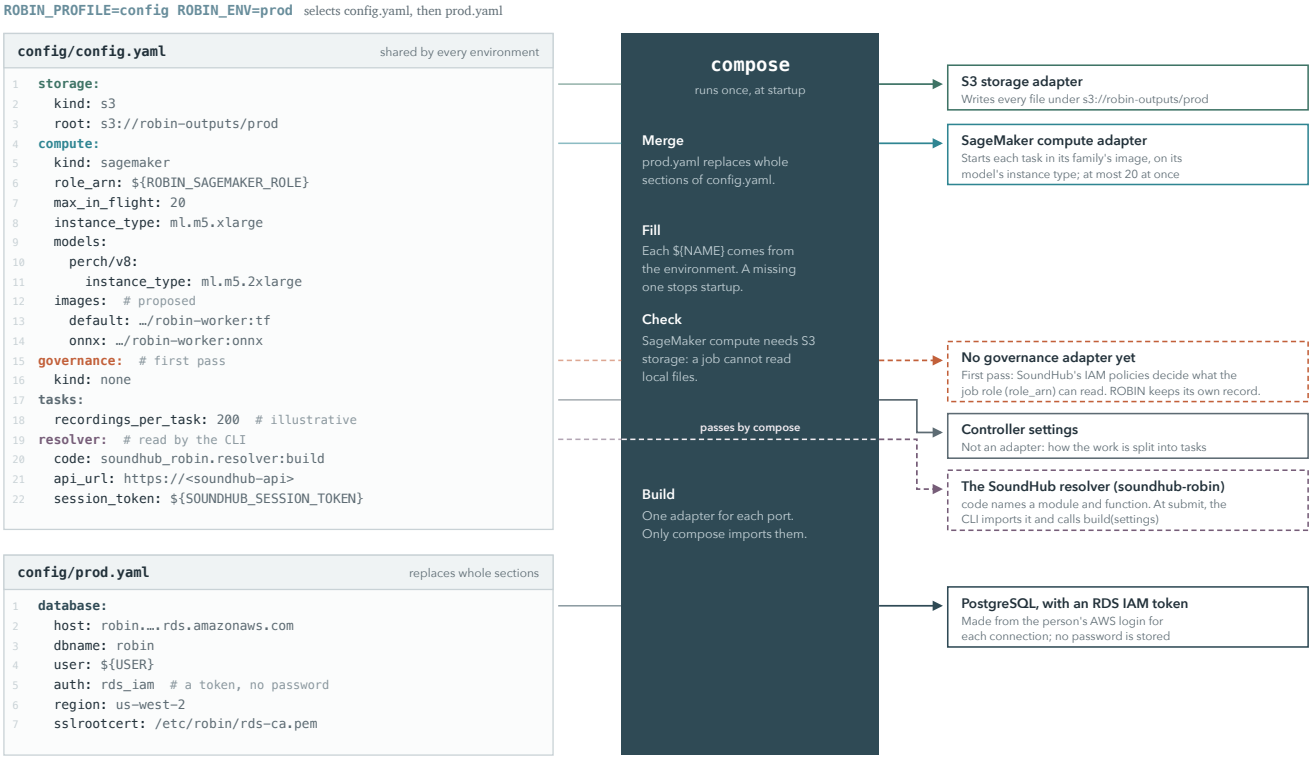


Figure 10 A deployment profile selects one adapter per port.

E. Run manager database design

The run manager keeps its records in a PostgreSQL database.

> **Note**
>
> PostgreSQL is the only database choice, either on a laptop or lab server (in a Docker container) or in a larger-scale production environment (Amazon RDS). There may be reasonable objections to this, arguing that the database should itself follow the port/adapter pattern, but this has been deferred for now.

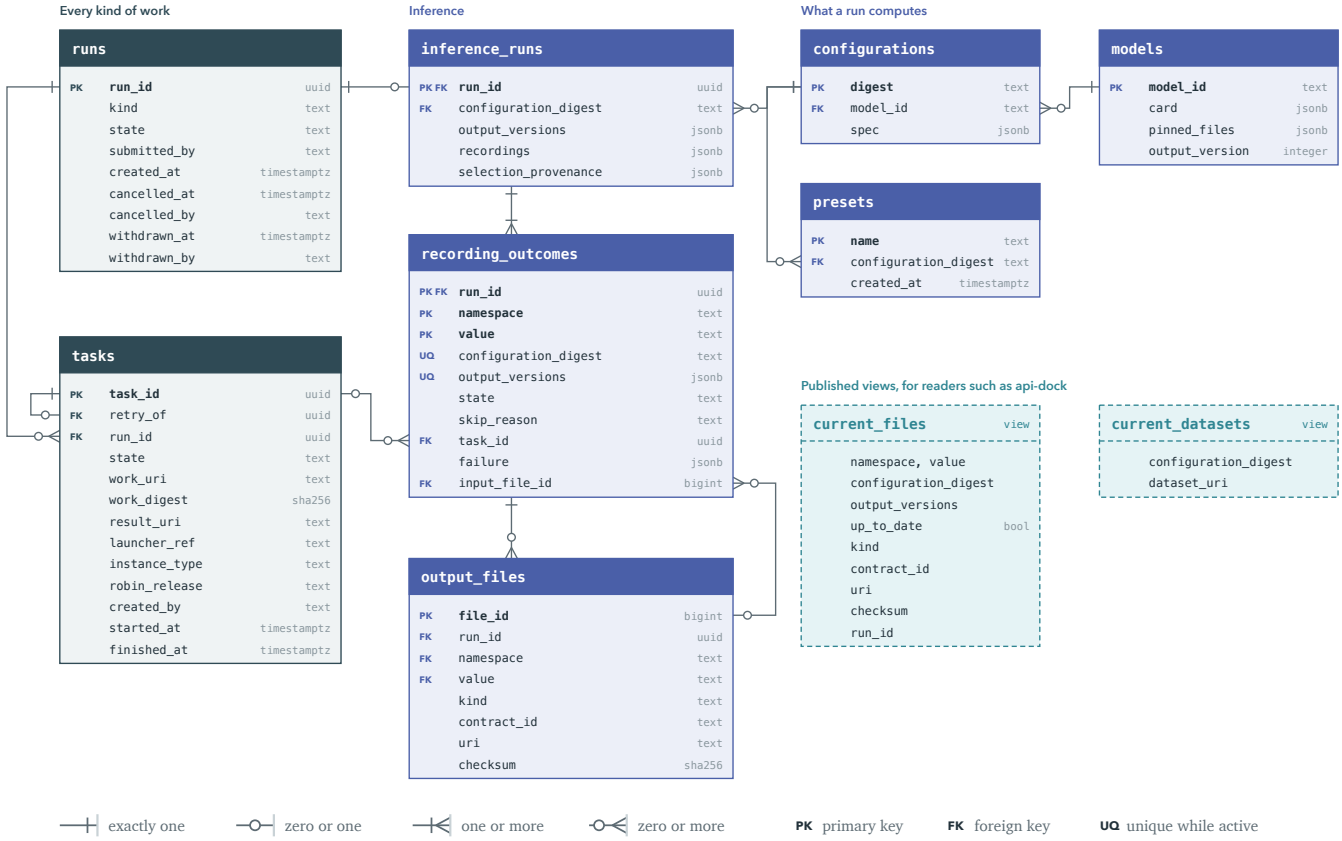


Figure 11 Tables in the run manager database.

Models, configurations, and presets

An operator registers a model from a YAML file that holds its card and pinned files (`robin models register FILE`). A model is identified by its name and version. Registering the same model again with the same card and files returns the existing row. The same name and version with different content is refused with `model_conflict`, because a changed model needs a new version.

A *configuration* is what a run computes, without its recordings: the model, what it reads (the recordings' audio, or the embeddings that a named backbone configuration saved), its settings, and the outputs requested. An operator writes it as YAML. ROBIN resolves it against the registered model and checks it with the same request checks the inference engine uses (they live in `robin_contracts`), so a bad configuration is refused when it is created, not later in a worker.

A configuration is identified by a SHA-256 digest of its resolved contents: a format tag, the model's name, version and card digest, the digest of each of the model's files, the input, the settings, and the output requests. Two runs with the same values have the same configuration, whether or not they came from a preset. File locations are left out, so moving a model's files does not create a new configuration. A test pins the digest of a fixed example, so a new release of ROBIN cannot silently change how configurations are identified.

A head's configuration names the backbone configuration whose embeddings it reads. ROBIN refuses it unless that backbone configuration saves embeddings, the head was trained on that backbone, and the embeddings are in the shape and format the head expects.

A *preset* is a fixed name for one configuration (`robin presets create NAME --configuration FILE`). A preset cannot be edited, and never needs to be: when a new release of ROBIN changes a model's outputs, it raises the model's output version (Output versions), which is not part of the configuration, so the preset still names the right thing. Creating the same preset again returns it. The same name for a different configuration is refused with `preset_conflict`.

No work is done twice

A recording has at most one valid result for each configuration and set of output versions. The database enforces this across all runs, including work that is still in progress: a unique index covers every recording outcome that is pending or covered, keyed by the recording (identified by a namespace and a value within it, such as a SoundHub recording ID), the configuration, and the output versions. So a double-click, a network retry, or two overlapping runs never compute the same thing twice.

ROBIN does not try to recognize a repeated request. Each request makes a new run, and the index stops the run from repeating work: when a run is prepared, it claims each of its recordings (Preparing a run), and a recording that is already covered or in progress elsewhere is recorded as skipped instead. A run whose recordings are all covered or in progress elsewhere finishes with no tasks.

Each recording in a run has one outcome. Only *pending* and *covered* hold a claim.

- **Skipped.** Already covered, or in progress in another run. The reason is recorded.
- **Pending.** Claimed by this run, and not yet finished.
- **Covered.** Finished, with its output files.
- **Failed.** Not finished, with the stage and reason. The claim is released, so another run can try again.
- **Released.** Given up by a cancel before the work started.
- **Withdrawn.** Removed from the current results by a withdrawal.

Current results

A published view, `current_files`, gives one row for each current file, by recording, configuration and kind: its location, its checksum, the run that made it, and whether it is up to date. A result is up to date when its output versions are the highest recorded for its model. Readers read exactly the files this view names, and never match paths with a pattern, because a pattern would also match withdrawn and failed runs.

F. The controller loop

The controller is the one process that makes sure that ROBIN's records in the run manager database match the state of the outside world. It prepares runs, launches tasks via the compute port, asks the compute port what happened, and records run outcomes.

ROBIN needs a control loop because a run spans two kinds of systems that may not necessarily be able to transact. The record is in a PostgreSQL database, and the work happens on a compute system (SageMaker, k8s, Slurm).

The design of the controller loop is derived from the following requirements:

- **Workers only exchange files.** They never touch the database and hold no database credentials, so they cannot record their own outcomes. The controller reads each result, checks it, and records it.
- **Outcomes come from asking, not being told.** Every compute adapter can answer "describe this job". Not every compute platform is capable of sending events to an external process. So the controller polls `describe` on the compute adapter, which updates the run manager on the status of the run.
- **Credentials belong to the machine the deployment is made from.** ROBIN doesn't store other people's credentials, because that creates a new origin for security vulnerabilities.

The cost of this design is latency. A result can wait up to one polling interval (30 to 60 seconds) before it is recorded, and the current dataset can be about a minute behind the database. For jobs that run for minutes or hours, and for systems without the capacity to send events back, this was accepted.

One pass

One pass of the controller does the following:

1. **Stop cancelled work and settle unconfirmed launches.** Ask the compute adapter to stop the started tasks of runs that are listed as cancelled or withdrawn in the database.
2. **Record ended tasks.** Ask the compute adapter for the state of each running task, and record the outcome of each one that has ended (see Recording an outcome).
3. **Prepare waiting runs.** Claim recordings and make tasks for new runs (see Preparing a run).
4. **Launch queued tasks**, oldest first, up to the limit of running tasks.
5. **Update the "current results" view** for the recordings whose results just changed.
6. **Wait for the next poll.**

Preparing a run

Submit stores the run with its frozen recording list and configuration. Preparation for a run happens in one transaction, which claims each recording with `INSERT ... ON CONFLICT DO NOTHING` against the unique index from No work is done twice. A **claim** is a pending outcome row that says "this run is processing this recording with this configuration". A recording whose claim fails is written as skipped, along with the reason. Finally it splits the claimed recordings into queued tasks and marks the run as ready.

Launching a task

A deployment profile's `MAX_IN_FLIGHT` caps the total number of tasks that can run at once, so a large run does not go over the compute system's quota. A task counts toward the limit if it is in the state `submitting`, `submitted`, or `running`.

Starting a task takes three steps, because the call to the compute system cannot be inside a single database transaction:

1. In a short transaction, check that the limit is not reached and that the task's output versions are current, and move the task from `queued` to `submitting`.
2. Outside the transaction, write `work.json`, and ask the compute adapter to start the job.
3. In a second transaction, save the job's ID and mark the task `submitted`.

Unconfirmed launches. The controller can stop between steps 2 and 3: the job may have started, but the database does not know it. Each job's name comes from its task's ID, and every compute adapter must make a repeated start of the same task return the job that already exists, not start a second one. So the next pass simply repeats step 2. If the adapter cannot tell what happened, the task keeps its place under the limit and ROBIN reports it as uncertain, rather than risk running it twice.

A task moves through these states:

- **Queued.** Waiting for a free place under the limit.
- **Submitting.** A place is reserved; the job may or may not have started.
- **Submitted.** The compute system has accepted the job.
- **Running.** The compute system reports that the job has started.
- **Finalizing.** The job has ended. ROBIN still has to read and record its result. The task no longer counts toward the limit.
- **Succeeded.** A valid result is recorded. Some recordings may still have failed.
- **Failed.** The launch, the whole work, or the result file failed.
- **Cancelled.** Cancelled before launch, or stopped without a result.

Recording an outcome

When the compute system reports that a job has ended, the task moves to `finalizing` and no longer counts toward the limit. The controller then reads `result.json` and checks three things:

1. It names the SHA-256 of the work file that the controller wrote.
2. It gives an outcome for every recording in the work.
3. Every output file it names is inside the task's folder.

Then, in one database transaction, the controller locks the task and records the task's outcome, each recording's outcome, and the output files. Recordings with valid results become current when the transaction commits.

Cancels, withdrawals, and retries

These commands change the database all at once. The controller does the outside work on its next pass. Preparing, starting, and recording each lock the run and read it again, so all three see a cancel or a withdrawal in the same order.

- **Cancel.** Frees the claims of queued tasks at once. The next pass stops the tasks that have started. If a valid result arrives after the cancel, ROBIN still records it.
- **Withdraw.** Does the same, and also takes the run's results out of the current results, in the same transaction. A result that arrives later never becomes current.
- **Retry.** Makes a new task for the run's failed recordings, with `retry_of` pointing to the old task.

Nothing is deleted, and each command is safe to repeat.

Running the controller

The controller runs via the CLI command `robin reconcile`. Without any options it does a single pass of the controller loop and exits. With `--loop` it will repeat passes until it gets a `SIGINT` or `SIGTERM` from the operating system. The controller loop is designed to be run on the web server that hosts the API. Until then, it should be run while runs submitted via the CLI have not concluded.

G. CLI design

Operators use the `robin` command from the CLI. It calls the run manager's operations and connects to PostgreSQL using an operator's specific credentials. Every command has the form:

```
robin [--profile DIR] [--env ENV] [--json] <noun> <verb> [arguments]
```

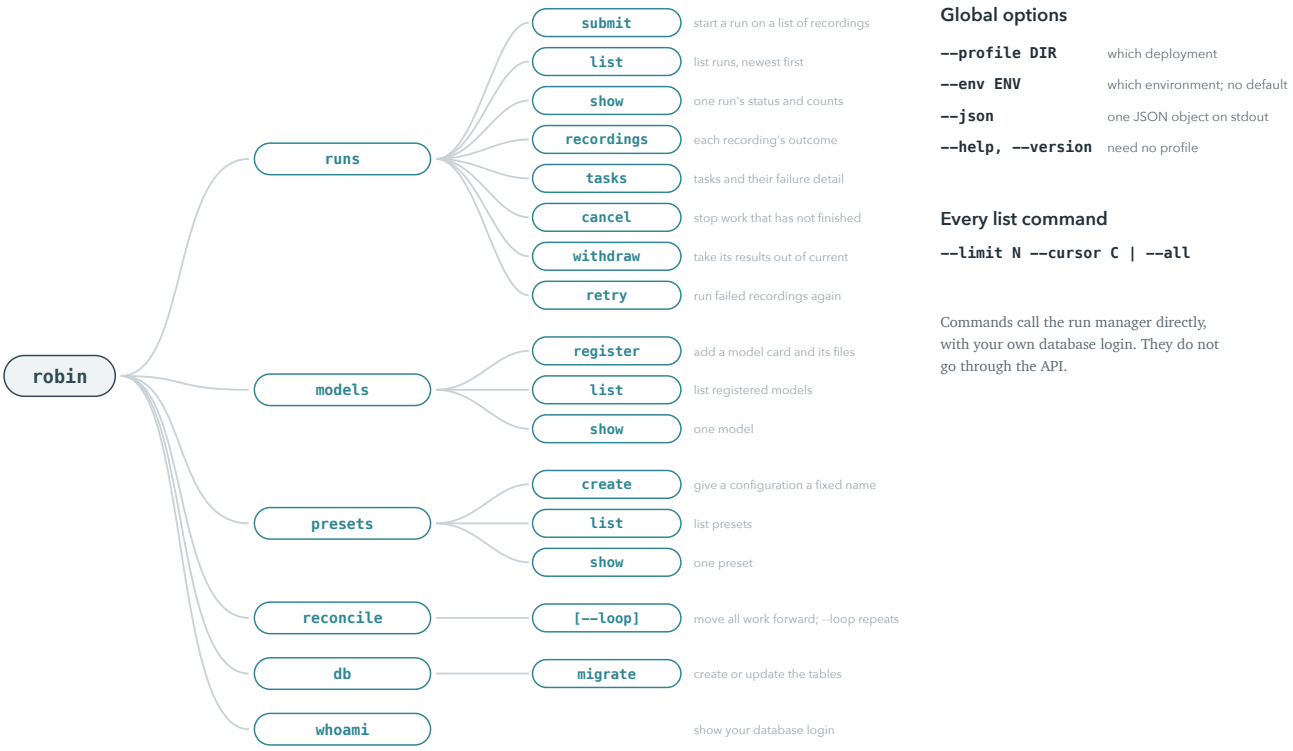


Figure 12 The `robin` command tree.

What happens when a command runs

Every command goes through the same seven steps before it does its own work. A failure at any step stops the command, and nothing changes in the database.

1. **Choose the deployment.** The profile folder comes from `--profile` or, failing that, `ROBIN_PROFILE`. The environment comes from `--env` or `ROBIN_ENV`. There is no default environment, and ROBIN never runs on `config.yaml` alone, so every command names the

deployment it acts on. If the folder, the environment, or the environment's file is missing, the command exits with code 2. `--help` and `--version` skip all of this.

2. **Load the profile** (Section D). The environment file replaces whole sections of `config.yaml`, then each `${VAR}` is filled in from the shell. An unset variable or a malformed file is refused with `invalid_profile`. From here on, every command prints the profile and environment it is using on stderr, so a stale `ROBIN_ENV` left in a shell is easy to spot.
3. **Test the login.** The CLI opens one connection to the database and closes it, so a failure has a precise reason: `not_authenticated` (the login was refused), `database_not_found`, `server_not_verified` (the server's certificate failed the check), or `unavailable`. The server's own error text is never shown, because it can contain the password.
4. **Open a connection pool.** The pool's size and time-outs come from the profile. With `auth: rds_iam`, each new connection gets a fresh IAM token, made on the operator's own computer from their AWS credentials.
5. **Ask the database who is running the command.** This is the *actor*: the name ROBIN records as responsible for an action, such as `registered_by` on a model or `submitted_by` on a run. The CLI does not take it from a flag or from the computer's user name. It asks PostgreSQL, with `SELECT session_user`, which returns the database login that PostgreSQL has just verified. Because the operator cannot simply type a name, a run is attributed to whoever's credentials actually logged in. This is a short read-only query, made once per command, before the schema check, so it works even on an empty database (which is why `robin whoami` works before `robin db migrate`).
6. **Check the schema.** Every command except `db migrate` and `whoami` is refused with `schema_not_current` unless the database has exactly this release's migrations: the same versions, names, and SHA-256 checksums. If the database is only behind, the message says to run `robin db migrate`. If it has different migrations, the message says to use the release that migrated it.
7. **Run the operation.** Each operation is one function that owns its own transaction. The CLI passes the actor from step 5 to it as an argument, and the operation writes it into the record. The database never fills in the actor by itself. This matters for the API: it will connect as one database login on behalf of many people, so `session_user` would name the API, not the person, and the API will pass its own actor instead. When the operation finishes, the CLI closes the pool.

The actor is also what ties a run to its submitter's credentials. `robin reconcile` only launches, stops, and records the outcomes of runs that the current actor submitted, so one operator's credentials are never used for another operator's run. Attribution is only as strong as the database logins behind it: each person needs their own login, and an IAM policy that lets a person log in only as themselves.

H. API design

The API gives web interfaces (like SoundHub) the same operations as the CLI. It is still in the planning phase. It will be ROBIN's own server: a standalone FastAPI application run by a given deployment. It supports the same operations as the CLI, with the exception of `robin reconcile`

(see Running the controller), which is intentionally run as a background process by the web server.

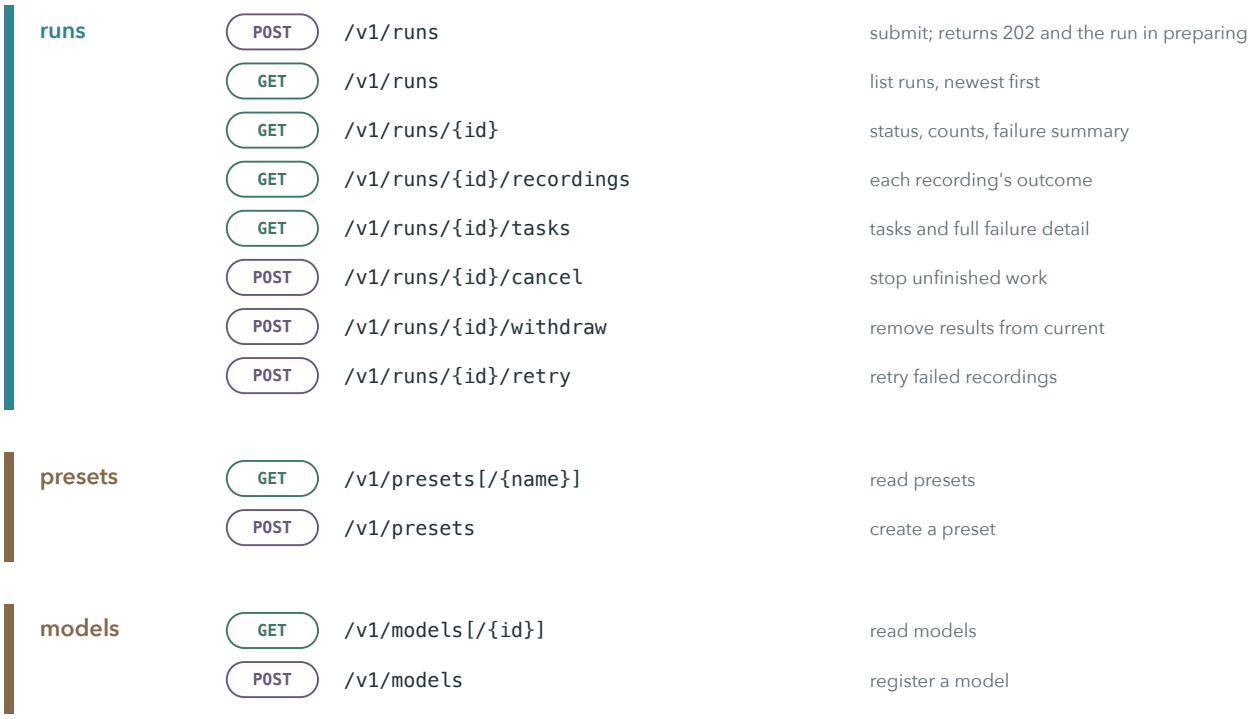


Figure 13 The planned API surface.