The Vivarium Users Guide

Compose living models.

bigraph-schema · process-bigraph · vivarium-workbench · viva-superpowers

vivarium-collective.github.io/viva-docs

Generated 2026-09-24

The Vivarium Users Guide

Compose living models.

Vivarium is a framework for building **multiscale biological models** by composing independently-written simulators into one executable whole — and for turning the runs into **auditable scientific evidence**. It is built as two spines wired into a single discovery loop:

⚙ The computational spine

What runs. Typed **Stores** hold state; **Processes** and **Steps** wired to them carry the dynamics; whole simulations nest as **Composites**. Updates are deltas merged by the type system.

🧠 The agentic spine

What reasons. Each run is a **Study** inside an **Investigation**. Studies form a gated DAG; a code-computed evaluator rolls runs up into a **verdict**, a readiness score, and open epistemic debts.

Neither spine is the engine — the engine is the closed loop between them, and the **Workbench** is where the loop turns.

The four packages

Each layer imports the ones below it and never the reverse. Click through to any layer.

viva-superpowers `the /viva-* skills`

The Claude Code skills that author and run everything. All AI lives here, so the tool below stays auditable.

vivarium-workbench `the dashboard server`

The AI-free server: investigations, studies, runs, analyses, report cards — every change committed to git.

process-bigraph `the engine`

The composite / process / step / template primitives, the tick scheduler, and the emitters that record state.

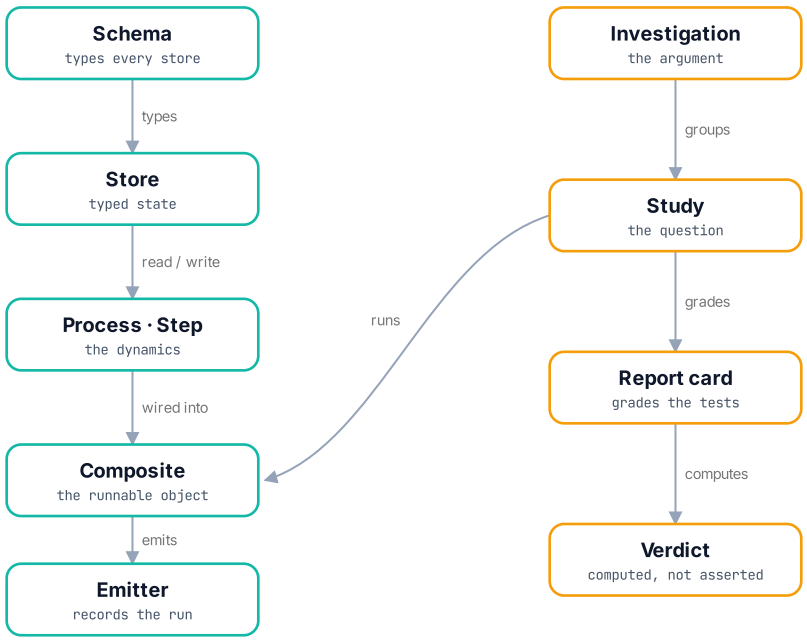
bigraph-schema `the type system`

The types beneath every store. `apply` is the one law that merges deltas, so independent processes compose.

> *A Composite is the runnable object; a Study is the reason you run it; an Investigation is the argument several studies build together.*

How it all fits together

Read the map bottom-up and it's a simulation; read it top-down and it's an argument. Every box links to the chapter that covers it.



Choose a path

🧭 New here?

Start with **What is Vivarium?** for the mental model, then **Core concepts** for the vocabulary.

🛠️ Build a model

Go to **Processes & Steps** and **Composites & wiring** to write and run your first composite.

Do science

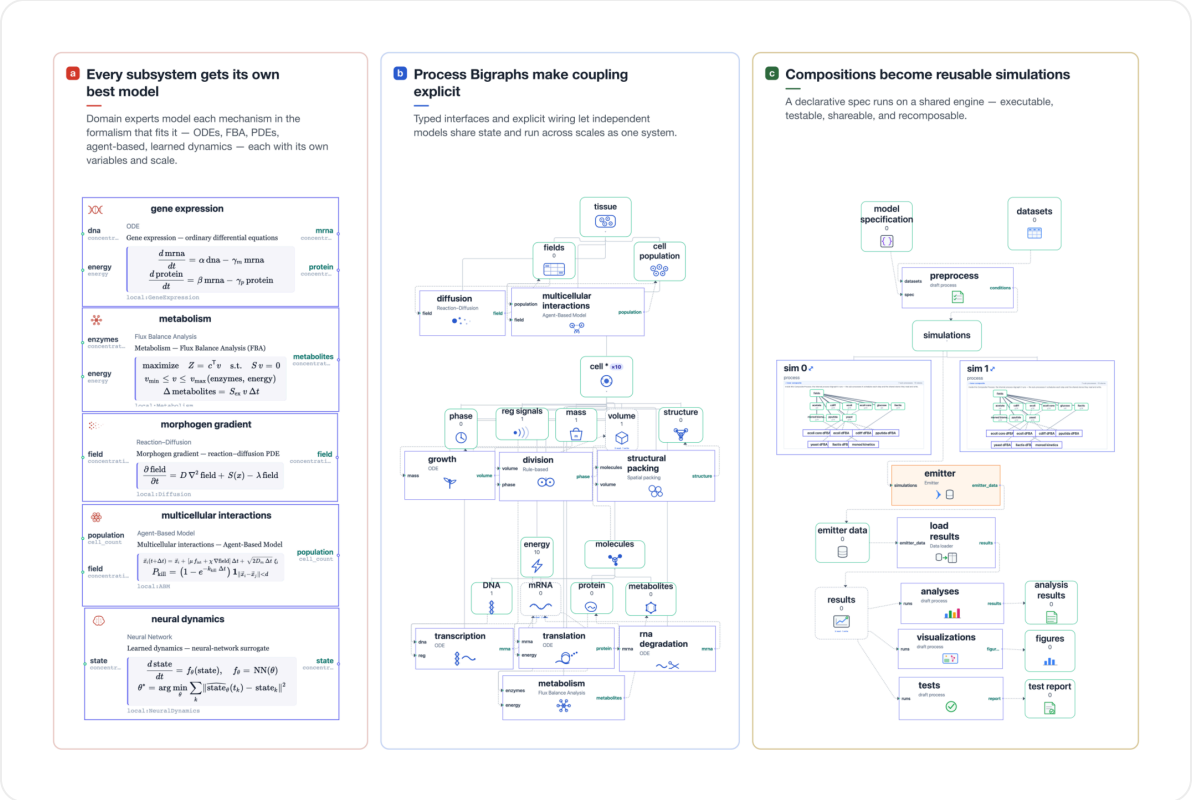
Head to **Studies** and **Investigations** to turn runs into gated, verdict-bearing evidence.

Look something up

The **Reference** section has the skill catalog, HTTP API, schemas, install/deploy, and a full worked example.

See the **Start here** guide for full step-by-step learning paths.

What a composed model looks like



Process Bigraph composes heterogeneous models into executable, reusable multiscale simulations. *(a) Each subsystem is modeled in the formalism best suited to it — ODEs, flux-balance analysis, reaction–diffusion PDEs, agent-based models, learned dynamics. (b) Typed* ***process interfaces*** *expose variables and make coupling explicit, connecting models across formalisms and scales. (c) Process Bigraph encodes components, interfaces, and couplings as declarative specifications that execute, analyze, test, and share.* (Agmon & Spangler, Fig 1.)

The Vivarium framework is described in Agmon & Spangler, *Process bigraphs and the architecture of compositional systems biology* (arXiv:2512.23754). Source packages live at github.com/vivarium-collective.

Start here

This guide is organized into four parts — Foundations, Build & run models, Investigate, and Reference. You don't have to read them in order. Pick the path that matches what you're trying to do; each is a short, numbered route through the chapters that matter for it.

Two things almost everything assumes

Most hands-on work needs **a workspace** (a directory with a `workspace.yaml` and a `viva_<pkg>/` package) and **a running Workbench** (the dashboard server). If either is missing, start at Workspaces & the Workbench.

Just want it running?

The **Quick start** is a copy-pasteable, top-to-bottom path from nothing to a running Workbench — and it's written so you can point an **AI agent** straight at it.

Everyone: the 20-minute mental model

Before any path, read these three — they're short and everything else builds on them.

1. **What is Vivarium?** — the Two Spines and the problem it solves.
2. **Core concepts** — stores, processes, steps, wiring, the `apply` law, and sites/fill/ground.
3. **The stack** — how the four packages layer.

🛠️ Path A — Build a model

You want to wrap simulators as processes and run a composite.

1. Types & state

Schemas, types & state — how state is typed and how deltas merge.

2. Processes & Steps

Processes & Steps — write the two kinds of edge.

3. Compose & run

Composites & wiring — assemble, wire, and run a composite.

4. Get data out

Emitters — record the run into a durable store.

5. Design interface-first

Templates & draft processes — sketch an interface before the mechanism.

Then: run it in the UI via Workspaces & the Workbench.

🧪 Path B — Turn runs into evidence

You want reproducible, inspectable, verdict-bearing science.

1. Set up

Workspaces & the Workbench — scaffold and start the server.

2. Ask a question

Studies — wrap one question, one emit-contract, one pass/fail bar around a composite.

3. See the result

Analyses, visualizations & report cards — figures, tables, and graded scorecards.

4. Build the argument

Investigations — group studies into a gated DAG.

5. Make it defensible

Rigor & the evidence engine — computed-not-asserted verdicts, gates, and provenance.

Then: follow it end to end in A worked example.

Path C — Drive it with agents

You want AI agents to build and run investigations on your behalf.

1. How agents fit

Working with AI agents — the AI-free tool + swappable-plugin split and the access contract.

2. The skills

Skill reference — every `/viva-*` command and what it wraps.

3. The API

HTTP API reference — the endpoints agents call.

4. The shapes

On-disk schema reference — the YAML shapes agents read and write.

> New science is a **new study**, not a patch to the model.

Whichever path you take, the loop is the same: **Question** → **Composite** → **Run** → **Verdict** → **Next.** When you're ready for the full detail, everything is in the Reference.

agents getting-started workbench workspace

Quick start

This is the fast path from nothing to a running **Vivarium Workbench** over a fresh workspace — written so a **human or an AI agent** can follow it top to bottom. For the full picture, see Workspaces & the Workbench and Working with AI agents.

Pointing an AI agent here?

Give the agent this page's URL, or the raw Markdown so it can read it verbatim: `https://raw.githubusercontent.com/vivarium-collective/viva-docs/main/docs/quickstart.md`. Everything below is copy-pasteable and assumes no prior state.

What the Workbench is: a local web UI + HTTP API over a *process-bigraph workspace* (a folder with a `workspace.yaml` holding composites, studies, investigations, and runs). It **reads and writes** the workspace's files and commits each change to git, so every action has an audit trail.

Two layers, kept separate:

- **The Workbench** = server / UI / data. It has **no AI dependencies** — pure Python + static assets.
- **The LLM layer** = the `viva-superpowers` Claude Code plugin (skills that *drive* the Workbench's HTTP API). All AI lives here, never in the Workbench (see §4).

---

0. Prerequisites

- **Python ≥ 3.11**, **git**, and **uv** (`pip install uv` if absent).
- Optional (for the AI layer): **Claude Code** + the `viva-superpowers` plugin.

---

1. Create the workspace repo (scaffold from `viva-template`)

A workspace is a self-contained research repo (its own Python package + specs). Don't hand-roll it — scaffold from the template:

```
# Option A: "Use this template" on github.com/vivarium-collective/viva-template,
then clone your new repo.
# Option B: clone the template directly:
git clone https://github.com/vivarium-collective/viva-template my-workspace
cd my-workspace

bash use-this-template-init.sh     # prompts for a workspace name; renders the
.j2 files
uv venv && source .venv/bin/activate
uv pip install -e ".[dev]"          # this pulls in vivarium-workbench as a
dependency
python3 scripts/lint-workspace.py   # expect: "workspace lint: OK"
```

After this you have: `workspace.yaml`, a `viva_<name>/` Python package (older templates use `pbg_<name>/`), `composites/`, `studies/`, `investigations/`, and a `.pbg/` control dir. The template works standalone — **no plugin or Claude Code required** for the base tool.

> **Adding to an existing project**
>
> If you're adding the Workbench to an *existing* process-bigraph project instead of scaffolding, ensure the repo root has a valid `workspace.yaml` and `uv pip install vivarium-workbench` into its venv. (During beta it may not be on PyPI — install editable from a clone of `vivarium-collective/vivarium-workbench` if the PyPI install fails.)

---

2. Launch the Workbench

From the workspace root (the dir containing `workspace.yaml`):

```
vivarium-workbench serve --workspace .            # picks a free port, renders
once, then serves
# or:  vwb serve --workspace . --port 8000 --host 0.0.0.0
```

CLI subcommands (via `vivarium-workbench` or `vwb`): `serve · run · study · investigation · composite · rerun · runs · status · logs`.

On start it writes the live base URL to `.pbg/server/server-info` — that file is how *everything* (including agents) discovers the server.

---

3. Verify it's up (agent-friendly checks)

`server-info` is a small JSON document (`{ "port", "host", "url", … }`) — read the `url` field rather than the raw file:

```
BASE=$(python3 -c "import json; print(json.load(open('.pbg/server/server-info'))['url'])")
curl -s "$BASE/api/workspace-manifest" | head    # one-call situational-awareness snapshot
```

Useful endpoints:

- `GET /api/workspace-manifest` — one call returns the workspace, composites, studies, registry, health, and skills. *Start here for orientation.*
- `GET /api/linkage-index` — the deterministic cross-reference / navigation graph (what links to what).
- `GET /openapi.json`, `/docs`, `/redoc` — the live, auto-generated API contract. Prefer `/openapi.json` over prose when you need exact request/response shapes.

---

4. The LLM layer — how an AI works with the Workbench

4.1 The principle

The **Workbench is AI-free**. All AI capability is packaged as the `viva-superpowers` Claude Code plugin: a set of `viva-*` **skills** that call the Workbench's HTTP API to author and run models. This keeps the tool auditable and the AI swappable.

4.2 One-time agent setup

```
1. Install the `viva-superpowers` plugin in Claude Code.
2. Run  /viva-init         — installs every viva-* skill into ~/.claude/skills/
(once per machine).
3. Run  /viva-workbench start  — launches the Workbench and writes
.pbg/server/server-info.
   (Skills read that file for the base URL; if it's absent they'll tell you to
start the server.)
```

4.3 How an agent talks to the Workbench (the access contract)

- **Base URL:** always read from `.pbg/server/server-info`. Never hardcode a port.
- **No token / no auth to fight:** the CSRF guard is **same-origin**, and a request with **no `Origin` header is always allowed** — so `curl` / `httpx` /CLI calls just work.
- **Read the contract, not the prose:** `GET /openapi.json` is authoritative for shapes.
- **Orient in one call:** `GET /api/workspace-manifest` (state) + `GET /api/linkage-index` (graph) before you start editing.
- **Runs are asynchronous:** a run returns a `run_id`; poll its status until done; the durable artifact is `studies/<slug>/runs.db`. Check the run's *result field*, not just the HTTP status (a failed run can still return 200).
- **Every write commits to git** in the workspace — your changes are a reviewable history.

4.4 The skills

A tour of what each `/viva-*` skill is for is in the **Skill reference** (some capabilities named in older docs — `viva-explore`, `viva-status`, `viva-cite-bands`, `viva-biology-forward` — are now subcommands of other skills; the reference lists the current set).

4.5 The authoring flow (typical agent arc)

```
scaffold/serve workspace  →  viva-catalog (get processes)  →  viva-expert (wrap
a new simulator, if needed)
   →  build a composite (compose processes)  →  viva-study (define a study over
that composite)
   →  viva-run (execute; poll to completion)  →  viva-viz / viva-report (render)
   →  viva-investigation (organize studies + narrative into a rigorous
investigation)
```

4.6 The mental model you're operating in

The framework collapses to **one object, one operation, one law**:

- **One object** — a *document*. A composite, a study, an investigation, and a template are the **same kind of thing** (a bigraph document).
- **One operation** — **fill**: substitute values / composites into a document's open **sites** (holes).
- **One law** — **groundness** (`is_ground`): a document runs iff it has no unfilled required sites.

Four features that look like separate machinery are that one idea in different places: optional members, gating on a failed prerequisite, pulling a cached result, and partial-graph triggering. See Core concepts and The stack for the full picture.

---

5. Quick reference

```
# --- one-time, new repo ---
git clone https://github.com/vivarium-collective/viva-template my-workspace &&
cd my-workspace
bash use-this-template-init.sh
uv venv && source .venv/bin/activate && uv pip install -e ".[dev]"
python3 scripts/lint-workspace.py           # -> "workspace lint: OK"

# --- launch ---
vwb serve --workspace .                      # writes .pbg/server/server-info

# --- agent orientation ---
BASE=$(python3 -c "import json; print(json.load(open('.pbg/server/server-info'))
['url'])")
curl -s "$BASE/api/workspace-manifest"       # state snapshot
curl -s "$BASE/api/linkage-index"            # navigation graph
curl -s "$BASE/openapi.json"                 # exact API shapes

# --- AI layer (Claude Code) ---
# /viva-init            (once per machine: install skills)
# /viva-workbench start (launch + write server-info)
# /viva-catalog /viva-study /viva-run /viva-expert /viva-investigation ...
```

Key files: `workspace.yaml` (workspace root marker) · `.pbg/server/server-info` (live base URL) · `studies/<slug>/runs.db` (durable run artifacts) · `composites/`, `studies/`, `investigations/` (the documents).

Golden rules for an agent: read `.pbg/server/server-info` for the URL; orient via `/api/workspace-manifest` first; trust `/openapi.json` for shapes; runs are async (poll, and check the *result*, not just HTTP 200); every write is a git commit.

PART 1

Foundations

The mental model and the vocabulary before any code. If Vivarium is new to you, start here: it explains what the framework is for, names every primitive from first principles, and shows how the pieces layer.

What is Vivarium?

The two spines, the rebuild tax it removes, and the discovery loop that makes it trustworthy.

Core concepts

The whole vocabulary from first principles — stores, processes, steps, wiring, apply, and sites/fill/ground.

The stack

How the four packages layer in strict dependency order, and why that acyclicity is the separation of concerns.

Suggested path

Read **What is Vivarium?** for the mental model, then **Core concepts** for the vocabulary everything else builds on, then **The stack** to see how the packages fit.

concepts getting-started

What is Vivarium?

Vivarium is a framework for **composing multiscale biological models** and for turning their runs into **auditable scientific evidence**. This chapter gives you the mental model everything else in the guide builds on.

On this page

Assumes no prerequisites — start here. · **You'll learn** why the rebuild tax exists, the two spines and the discovery loop between them, and what makes a study's conclusion trustworthy.

The problem: the rebuild tax

Biological systems are modeled with wildly different formalisms. Gene expression is naturally a set of ODEs; metabolism is a flux-balance (FBA) linear program; a morphogen gradient is a PDE; a growing colony is an agent-based model. Each of these is written in its own tool, with its own assumptions, units, and time scales.

When you want a model that spans scales — say, a whole cell whose metabolism feeds its gene expression, inside an environment with diffusing nutrients — you normally pay the **rebuild tax**: you reinterpret the literature, reconstruct the model structure, and re-implement everything in one monolithic simulator. Every new question means editing the monolith, and every edit risks breaking the parts that already worked.

Vivarium removes that tax by making models **compose**. Independently-written simulators are wrapped as typed **Processes** and wired together through explicit, checkable interfaces. A whole cell becomes a single named composite you assemble from parts; adding or swapping a subsystem is *wiring*, not forking. It's a shift in stance: you work less as a modeler rebuilding biology from scratch than as a

meta-modeler, wiring existing models together and letting the biological meaning emerge from how they are coupled and orchestrated across scales.

> **The guiding principle**
>
> New science is a **new study**, not a patch to the model.

The two spines

Vivarium is built as **two spines wired into one discovery loop**.

⚙ The computational spine — *what runs*

A compositional runtime built on Milner's bigraphs. State lives in typed **Stores**; dynamics live in **Processes** (temporal) and **Steps** (reactive) wired to those stores; whole simulations nest as **Composites**. Updates are **deltas** merged by the type system. Packages: `process-bigraph`, `bigraph-schema`, `viva-emitters`.

🧠 The agentic spine — *what reasons*

A knowledge layer that frames each model run as a **Study** inside an **Investigation**. Studies form a DAG through prerequisite gates; a code-computed evaluator rolls runs up into a **verdict**, a **readiness** score, and open **epistemic debts**. Packages: `viva-superpowers`, the study spine, the `/viva-*` skills.

A single simulation answers nothing on its own. The computational spine makes a model *run*; the agentic spine gives that run a *scientific shape* — a question it answers, a pass/fail bar it must clear, and a place in a larger argument.

> Neither spine is the engine — the engine is the closed loop between them, and the **Workbench** is where the loop turns.

The discovery loop

The two spines meet in a loop that runs at every scale of the work:

A **question** is written as a study; it points at a **composite** and runs it; the run emits a **run store** of trajectories; a code-computed **verdict** grades the run against the study's tests; and the verdict tells you what to ask **next**. AI agents can ride this loop at every hop — proposing studies, running composites, hardening findings, reading verdicts to choose what to ask next — while every step stays legible to a human on the study page.

What makes it trustworthy

Two design commitments run through the whole framework, and they are worth internalizing before anything else:

1. **A study's conclusion is computed from its evidence, not asserted.** You do not write `status: pass` by hand. The verdict is derived from the latest run's measured outcomes. This is what stops a study from *claiming* a result it never produced.
2. **The tool is AI-free; the AI is a swappable plugin.** The Workbench server, its data, and its evidence rendering contain no AI. All AI capability is packaged as the `viva-superpowers` Claude Code plugin — a set of `/viva-*` skills that call the Workbench's HTTP API. This keeps the tool auditable and the AI replaceable.

Because a study is **typed data**, hardening it — filling gaps, citing evidence, tightening a claim — is a transformation an agent can apply and a human can verify, not a matter of taste.

Who this guide is for

- **Modelers** who want to build a multiscale model without paying the rebuild tax → start with Core concepts, then Build & run models.
- **Scientists** who want reproducible, inspectable computational evidence → head to Studies and Investigations.
- **Tool builders and agent authors** driving the platform programmatically → see Working with AI agents and the HTTP API reference.

The foundational paper

The framework is described in **Agmon & Spangler, *Process bigraphs and the architecture of compositional systems biology*** (arXiv:2512.23754), which introduces **Vivarium 2.0** as the open-source implementation across three libraries — `bigraph-schema`, `process-bigraph`, and `bigraph-viz` — demonstrated with **Spatio-Flux**, a library of microbial-ecosystem simulations combining kinetic equations, dynamic FBA, and spatial processes. Its move is to take the architectural ideas that once lived inside the original Vivarium software and generalize them into a shared specification — of process interfaces, hierarchies, composition patterns, and orchestration — so models can be understood, reused, and built on rather than rebuilt.

Next: Core concepts — the vocabulary, from first principles.

composite concepts process step

Core concepts

This chapter builds the whole framework from first principles. The vocabulary is deliberately small: a handful of structural ideas, then a handful of temporal ones, then the way models compose. Everything later in the guide is an application of what is here.

The formal anchors come from the **Process Bigraph** paper's core vocabulary (Table 1); the plain-language framing comes from the *Composition Interface Protocol* primer. Where a term has a legacy synonym, it is noted — the ecosystem is mid-migration and older documents use different words for the same idea.

On this page

Assumes you've read What is Vivarium?. · **You'll learn** stores, edges and wires; processes vs steps; the delta-merge semantic that makes models compose; and how composites, templates and studies are all one kind of typed document.

Structure: stores, edges, and wires

At its base, a Vivarium model is a **bigraph** — a structure with two orthogonal parts: a **place graph** (what contains what) and a **link graph** (what is wired to what).

Stores — the state

A **Store** holds a typed value: a molecule count, a concentration field, the clock. If you think of a model as a program, stores are its **variables**. Stores nest hierarchically — a cell store contains a cytoplasm store contains a metabolites store — and that nesting *is* the place graph. A value's address is the **path** to it, e.g. `['cell', 'cytoplasm', 'glucose']`.

Formally, **state** is a map from paths to values, `x : P → V`, and a **schema** is a map from paths to types, `Σ : P → T` — "the typed organizational blueprint of the system." The separation between schema and state is the central principle: it lets you validate a model, reuse its structure, and reason about composition **independently of execution**.

Store diagrams: a single typed store, and stores nested in a place-graph hierarchy.

Store diagrams. *(a) A store is a rounded rectangle holding any data type, showing its name, value, and type. (b) Stores nest in hierarchies for multiscale representation — a place graph of nested stores (e.g. a cell containing cytoplasm, membrane, and nucleus).*

(Fig 3.)

Edges — the functionality

An **Edge** is a unit of functionality attached to stores. Each input or output of an edge is a **Port**. An edge never talks to another edge directly; it reads and writes **shared stores**, and that shared wiring *is* the coupling between them.

There are exactly two kinds of edge, and the difference is **time** (see below): a **Process** and a **Step**.

Wires — the coupling

A **Wire** connects a port of an edge to a store by its path, `W : (process, port) → P`. Wiring makes every coupling explicit and checkable. Where the place graph says *where* things are, wires say *how* they are connected.

Place graph vs link graph

- **Place graph** = containment = dict nesting. "Cell contains cytoplasm."
- **Link graph** = wiring = ports connected to store paths. "Transcription reads the gene store and writes the mRNA store."

Composition framework overview: Milner bigraphs (link graph + place graph) and process bigraphs (place graph + process graph).

Composition framework overview. *(a) Milner's original bigraphs combine a link graph (hyperedges, dashed) with a place graph (solid containment edges) over a set of nodes. (b) Process bigraphs replace the link graph with a* ***process graph*** *— processes connect to nodes through their typed ports.* *(Agmon & Spangler, Fig 2.)*

Time: processes and steps

The *Composition Interface Protocol* describes structure. What it does not specify is **time**. Process Bigraph adds a global clock and splits edges by their relationship to it. It helps to keep two things apart: *composition* is how edges are wired together through shared stores, while *orchestration* is when each one acts and how its updates combine.

A **Process** — temporal

Declares an **interval** (a timestep) and an `update(state, interval)` method that advances dynamics over a span of time — an ODE integrator, an FBA solve, a stochastic step, a spatial update. The clock drives it.

A **Step** — reactive

Declares **no interval**. Its `update(state)` fires when its inputs are ready — a dataflow rule run to convergence. Steps form a DAG that settles whenever the state they depend on changes. **Emitters, analyses, visualizations, and report cards are all Steps.**

Together, processes and steps let one simulation express both **continuous dynamics** (processes running at regular intervals) and **discrete events** (steps firing in response to conditions).

The one semantic that makes it compose

Here is the single most important idea in the framework:

> Processes never mutate state directly. Each `update()` returns a typed **delta**; the runtime merges it into shared state through the schema's `apply` method.

A process reads the state it is wired to, computes, and **returns a change** — it does not write anything itself. The runtime collects every process's delta and applies it through the type's own combination rule:

- **Accumulate** — numeric types add (two processes both consuming glucose subtract their amounts; the pool ends up correct).
- **Set** — replace the value.
- **Merge** — dict update.

Because *how deltas combine is a property of the data type, not the process*, two independently-written processes can write to the same store without knowing about each other. Numerical updates, structural rewrites, and scheduling therefore all live under one execution protocol. **This is what lets independently-written processes be wired together.**

Types and the type system

A **type** knows how to do a few things: produce a **default** value, **serialize** and deserialize, **check** whether a value is valid, and — crucially — **apply** a delta. These behaviors live in `bigraph-schema`. Types compose (a `map[string, float]`, an `array[(3|4), float]`, a `tree[float]`), carry **units**, and can declare inheritance. A composite's wiring is therefore *checkable, not ad hoc*: the type system knows whether two ports can legally connect.

The operational object is a **Core** — a registry of types (and of process classes) that the engine consults. You will meet it directly in Schemas, types & state.

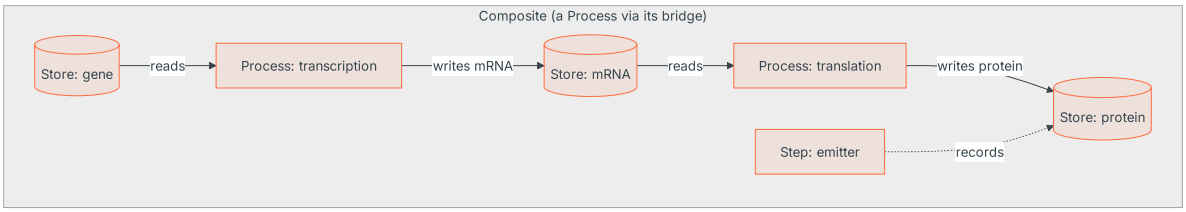
> ⚠ **Accuracy note**
>
> `bigraph-schema` was rewritten (v1.4.x). If you find older tutorials referring to a `TypeSystem` class or dict-keyed `_apply` / `_serialize` methods, that API is gone — the current object is `Core`, built via `allocate_core()`, with type behaviors dispatched as methods. The *concepts* above are stable; verify *code* against the current package.

Composition: the composite

A **Composite** is a state-tree of typed process and step nodes wired to shared stores. **It is the only object the engine actually runs.** Everything higher up in the framework merely points at a composite and records what came out.

A composite is a **document**: state + processes + the port wiring between them. Crucially, a composite is *itself a Process* — it owns an internal state-tree and scheduler and exposes a **bridge** that maps its external ports onto internal store paths. So a composite drops into a parent composite as a single node. This is how multiscale models are built: **containment, not hole-filling** — big models are

assembled from small ones. And because the pieces take on their biological meaning from *how* they are composed, the same processes wired into different patterns — cell–environment coupling, growth, division — give rise to different biology.



From documents to templates: sites, fill, and ground

A composite you can run is **ground** — every required slot is filled. A composite with a **hole** in it is a **template**.

- A **site** is a place-graph hole: a slot where a whole composite, process, or value plugs in. (This is Milner's bigraph *site*.) A composite **refuses to run** while any required site is open — "a hole is where a process should be."
- The one operation on documents is **fill**: substitute a value or a sub-composite into a site.
- The one law is **groundness** (`is_ground`): *a document runs iff it has no unfilled required sites.*

This collapses a lot of apparent machinery into one idea:

> One object (a typed document), one operation (**fill** its sites), one law (`is_ground`). A composite, a template, a study, and an investigation are all the same kind of thing — they differ in structure, not in execution machinery.

> **Legacy vocabulary**
>
> Older design documents call sites **slots**, and call filling them **bind** or **reify**. Prefer **site / fill / ground**; treat *slot* / *bind* / *reify* as synonyms when you meet them.

Draft processes

A **draft process** is a template idea applied to a single process. A `DraftProcess` declares a **contract** — its input and output ports plus a human-readable description of the transformation it is *meant* to perform — but carries **no dynamics**. Stepped, it stays inert and never fabricates behavior; it appears in the dashboard marked DRAFT. This lets you design an interface first and supply the mechanism later. See Templates & draft processes.

The knowledge concepts

Above the composite sit the concepts of the agentic spine. They are all **metadata that references a composite by a dotted id and records what came out** — none of them is the thing that runs.

Concept	One-line definition
Composite	The runnable object — the only thing the engine executes.
Study	The reason you run it: one question, one emit-contract, one pass/fail bar wrapped around a composite.
Investigation	The argument several studies build together: a gated DAG under one research question.
Run	One execution of a composite; produces a run store of trajectories.
Emitter	A Step that records wired state each tick into a durable sink.
Analysis / Visualization	Steps that read a run's normalized results and write artifacts (figures, tables).
Behavior test	A machine-checkable spec: how to *measure* a result and what *passing* means.
Report card	A Step that reads a run and produces pass/fail outcomes keyed by test — the raw evidence.
Acceptance band	The tolerance or direction a test accepts (e.g. "within 2%"), ideally cited to literature.
Finding	A summary of what was seen, floored by a lifecycle so an unproven claim can't look settled.
Verdict	The study's computed conclusion state (passed / failed / needs_calibration / blocked).
Provenance	Every write is a git commit; verdict, readiness, and debts are diff-able.

> **The sentence to remember**
>
> A **Composite** is the runnable object; a **Study** is the reason you run it; an **Investigation** is the argument several studies build together.

Each of these gets a full chapter in Investigate.

Next: The stack — how the four packages layer and depend on each other.

concepts getting-started

The stack

Vivarium is four packages in a strict dependency order. Each layer imports the ones below it and never the reverse. That acyclicity *is* the separation of concerns — and it is why you can swap the AI layer without touching the engine, or run the engine with no dashboard at all. At its heart, `process-bigraph` is a *composition protocol*: rather than unifying models into one representation the way standards like SBML or CellML do, it standardizes how independently-built models connect, share state, and are orchestrated in time.

On this page

Assumes you've read Core concepts. · **You'll learn** the four packages and their dependency order, how to read the stack bottom-up and top-down, which layer to reach for, and how the pbg → viva rename maps onto names you'll see.

4 · `viva-superpowers` — the `/viva-*` Claude Code skills + study/investigation authoring. Scaffolds workspaces, wraps simulators, authors and hardens studies, computes a deterministic rigor scorecard. **All AI in the ecosystem lives here.**

3 · `vivarium-workbench` — the dashboard server + the investigation-as-composite substrate. A FastAPI app that exposes the HTTP API, renders the study/investigation UI, runs composites, and commits every change to git. **Contains no AI.**

> **2 · `process-bigraph`** — the engine + the composite / process / step / template primitives. The `Composite` object, the tick scheduler, per-process intervals, and the emitters that record state.

> **1 · `bigraph-schema`** — the type system beneath every store. Types, ports, schema resolution, and the store-write law (`apply`). The `Core` registry that everything above consults.

One-line version:

> `bigraph-schema` (the type system) ⊂ `process-bigraph` (the engine + composite/template primitives) ⊂ `vivarium-workbench` (the dashboard server) ⊂ `viva-superpowers` (the `/viva-*` skills).

Read it two ways

Bottom-up, it's a simulation. Types define what state can be; processes and steps act on that state; composites assemble them; the engine runs them; emitters record them.

Top-down, it's an argument. A skill authors an investigation; the investigation groups studies; each study wraps a question around a composite; the composite runs; the evidence rolls back up into a verdict.

The whole point of the split is leverage:

> You can re-run the science without rewriting the argument — and audit the argument without re-reading the code.

What each layer gives you

Layer	You reach for it when…	Key objects
bigraph-schema	you need a new data type, units, or to understand how deltas merge	`Core`, types, `apply`, `resolve`
process-bigraph	you are writing or wiring a model	`Process`, `Step`, `Composite`, emitters
vivarium-workbench	you are running, browsing, or grading models in the UI/API	the server, `/api/*`, studies, report cards
viva-superpowers	you are driving the platform with an AI agent	the `/viva-*` skills

The unifying idea

The layers look like different kinds of machinery, but underneath they are one idea applied at different altitudes. Every Workbench object — a composite, a study, an investigation, a template — is a **`bigraph-schema`-typed document**. Once its sites are filled, it is a `process-bigraph` composite: Steps transform Stores, wiring expresses dependencies, and the engine produces durable artifacts.

- A **process** is a ground document.
- A **composite** is a document whose sites were filled by other documents — so a composite *is* a process; composition is closed.
- A **template** is a not-yet-ground document.
- A **study** is a template with its model site filled.
- An **investigation** is a document with one site per study, filled by gate edges.

Three slogans capture the design ethos this produces:

> **Declare, do not script. Compose, do not duplicate. Artifacts are outputs.**

You do not write orchestration code that runs studies in the right order — you declare each study's prerequisites, and *the wiring is the orchestration*: the engine schedules the network. Templates, studies, and investigations differ in **structure, not in execution machinery**.

> **Design vs shipped**
>
> The "everything is one typed document" unification is the framework's north star, and the load-bearing pieces have shipped (an investigation compiles to a composite; `bigraph-schema` has `fill`/`is_ground`). Some deeper unification is still in progress. This guide flags, per chapter, where a described capability is a current API versus a documented direction.

A note on names

The ecosystem was renamed from **pbg** ("process-bigraph") to **viva**. The migration is real but not complete, so you will see both:

- Skills are `/viva-*` (older `/pbg-*` names still work as aliases).
- Python **import names are stable**: `import process_bigraph`, `import bigraph_schema`.
- Some packaging still uses the old prefix (the plugin is distributed as `pbg-superpowers`; the runtime control directory is `.pbg/`).
- Newer workspaces use a `viva_<pkg>/` Python package; older ones use `pbg_<pkg>/`.

When in doubt, prefer the **viva** spelling; this guide notes the exceptions where they matter.

Next: start building with Schemas, types & state, or jump to Processes & Steps to write your first model.

PART 2

Build & run models

The computational spine — how you write and run composites. This part is for modelers: it turns the vocabulary from Foundations into the concrete API for typing state, writing edges, wiring them into a runnable whole, and getting the data back out.

Schemas, types & state

The `bigraph-schema` type system — what state can be and how deltas merge back into it.

Processes & Steps

The two kinds of edge, split by their relationship to time, from base-class contract to runnable code.

Composites & wiring

Assemble edges and stores into the one object the engine runs, and see what happens on every tick.

Emitters

The one part of a composite allowed to leave the run — recording wired state each tick to a durable sink.

Templates & draft processes

Interface-first modeling — design a document with the mechanism left out, then fill it in later.

Explore a bigraph

See a real composite in the live loom explorer — pan, zoom, and drill into its wiring.

Suggested path

Read in order: **Schemas** for the ground rules, **Processes & Steps** to write functionality, **Composites & wiring** to run it, then **Emitters** to capture the output. Reach for **Templates & draft processes** when you want to design an interface before its mechanism exists.

schema state types

Schemas, types & state

The `bigraph-schema` layer sits at the bottom of the stack. It answers one question before anything runs: **what can state be, and how do changes to it combine?** Every Store in a Vivarium model holds a value that this layer has typed, and every delta a Process returns is merged back into that state by a rule this layer owns. Get the type system right and independently-written simulators compose without knowing about each other; get it wrong and nothing above it can be trusted.

This chapter is the working reference for that layer. It builds on the vocabulary from Core concepts — stores, schema/state separation, deltas, `apply`, and the site/fill/ground story — and turns it into the concrete API you call. If you have not read Core concepts, start there; this chapter assumes you know *why* deltas merge and focuses on *how* the machinery works.

On this page

Assumes Core concepts. · **You'll learn** the type-string grammar, the `Core` registry, how `apply` merges deltas, and why schema and state stay separate.

A **schema** is a map from paths to types; a **state** is a map from paths to values. The schema is the typed blueprint; the state is one thing that inhabits it. Keeping the two separate is what lets you validate, reuse, and reason about a model independently of running it.

What `bigraph-schema` is

`bigraph-schema` "provides a serializable type schema for compositional and multiscale modeling … the foundation of the Vivarium 2.0 simulation framework." Concretely it is three things wearing one coat:

A type language

A compact string grammar — `float`, `map[float]`, `array[(3|4),float]`, `link[x:int|y:string]` — plus a set of base types (scalars, containers, wiring types, Milner's structural types) that the grammar composes.

A registry

The `Core` object: a registry of types, edges (links), and methods that the engine consults. You register a new type or process class once and every operation knows about it.

A reversible codec + algebra

Type-aware operations — `check`, `default`, `serialize` / `realize`, `apply`, `resolve` / `reconcile`, `traverse` — that all dispatch on the compiled type, translating between a compiled (in-memory) and an encoded (JSON-compatible) representation.

Its place in the stack is to supply **Schema**, **State**, and **Update** and the algebra that relates them; `process-bigraph` builds the Composite, Process, and Step on top:

Layer	What it is	Key operations
Schema `S`	a tree of type declarations	`check`, `default`, `realize`
State `X(S)`	a value inhabiting a schema	built from defaults; mutated by `apply`
Update `U(S)`	a value representing a change	combined by `reconcile`; consumed by `apply`
Process / Edge `P`	reads state, emits updates	invoked per-tick
Composite `C`	state containing processes	one tick = read → reconcile → apply

> ⚠ **Accuracy note — this is the rewritten API**
>
> `bigraph-schema` was substantially rewritten (this guide tracks **v1.4.x**). If you find older tutorials referring to a `TypeSystem` class, a single `type_system.py`, or type definitions as dicts of string-keyed `_apply` / `_serialize` / `_check` methods — **that API is gone.** There is no `TypeSystem` class in this tree. Today:
>
> - Base types are Python **dataclasses** subclassing `Node`.
> - Type behaviors are `plum`-dispatched **multimethods**, one module per operation under `bigraph_schema/methods/`, each specializing on the dataclass type — not string keys on a type dict.
> - The operational front door is the `Core` object, built with `allocate_core()`.
>
> The `_`-prefixed names survive only as (a) dataclass **schema fields** (`_type`, `_default`, `_value`, `_inputs`, …) and (b) on-the-wire **update sentinels** (`_add`, `_remove`, `_divide`) that the `apply` router dispatches to method modules. The *concepts* from Core concepts are stable; verify *code* against the current package.

A note on the name

Unlike most of the ecosystem, this package was **not** renamed in the pbg→viva sweep. The import is `import bigraph_schema`, the PyPI distribution is `bigraph-schema`, and the repo stays at `vivarium-collective/bigraph-schema` — it is kept under its upstream name as a foundation. See A note on names.

The `Core` object

`Core` is a registry plus the operations that consult it. You almost always build one with `allocate_core()`:

```
from bigraph_schema import allocate_core

core = allocate_core()      # base types + all discovered packages, as an
isolated copy
```

`allocate_core()` builds a base `Core` seeded with the `BASE_TYPES` registry, runs package discovery (it walks installed distributions and auto-registers any types, edges, or visualizers they advertise), caches that, and hands back an **isolated copy**. Isolation is the point: registering a type, link, or method on one core never leaks into another.

`core.registry`

string key → type (a `Node` class, instance, or dict). The type namespace.

`core.link_registry`

string key → edge class. Where process/step classes live so a `link` can name one.

`core.method_registry`

string key → callable. Extension hooks the operations can call by name.

```python
def test_allocate_core_isolation():
    a = allocate_core()
    b = allocate_core()

    # Mutable containers and caches are distinct objects per instance.
    assert a.registry is not b.registry
    assert a.link_registry is not b.link_registry

    # Both copies carry the shared base types.
    assert 'float' in a.registry and 'float' in b.registry

    # Mutating one must not leak into the other.
    a.register_type('zzz_isolation_probe', 'float')
    assert 'zzz_isolation_probe' in a.registry
    assert 'zzz_isolation_probe' not in b.registry
```

Abridged from `tests.py::test_allocate_core_isolation` (which also asserts `method_registry` and the internal caches are distinct, and probes link/method isolation).

Why isolation matters

A study, a composite, and a scratch experiment can each hold their own `Core` and register bespoke types without stepping on one another. When you need the shared base plus package-discovered types, use `allocate_core()`. When you want *only* the base types with no discovery (tests, tight control), you can construct `Core(BASE_TYPES)` directly — but `allocate_core()` is the normal path.

Registering types, links, and methods

```
core.register_type('my_conc', 'map[float]')        # any schema form is accepted
core.register_link('my_process', MyProcessClass)   # an Edge subclass
core.register_method('m', lambda core, *a, **k: ...)
```

`register_type(key, data)` accepts a string expression, a dict, or an already-compiled `Node`. If the key already exists it **deep-merges** the new definition over the old one (via `resolve`, below) rather than clobbering it — so a package can extend a type another package registered. `register_types`, `register_links`, and `register_method` are the bulk / edge / hook variants.

Three ways to declare a schema

A schema is a type declaration, and you can write it three ways. All three normalize to the same compiled `Node` through `core.access()` — which is the single funnel every operation passes its schema argument through.

1 · String DSL

A compact expression parsed by a `parsimonious` grammar:

```
core.access('float')
core.access('map[float]')
core.access('array[(3|4),float]')
core.access('link[x:integer,y:string]')
core.access('tree[float]')
```

The grammar composes types:

Syntax	Meaning
`foo[A,B,C]`	parameterize — `A,B,C` fill the type's schema fields in order
`a|b|c`	**merge** — dicts merge into one mapping; non-dicts form a `Tuple`
`a~b~c`	**union** — becomes `Union(_options=[…])`
`key:subtype`	**nest** — becomes `{key: subtype}`
`(…)`	group
`type{default}`	default block — `float{5.5}`, `string{hello}`, `integer{11}`

So `array[(3|4),float]` is an array whose shape parameter is the merge `(3|4)` and whose data type is `float`; `link[x:integer,y:string]` is an edge with two typed input ports.

2 · Dict

A dict with underscore-prefixed **schema keys** (`_type`, `_default`, `_key`, …), and/or a structured dict of named sub-schemas:

```
node_schema = {
    'a': {'_type': 'float',  '_default': 11.111},
    'b': {'_type': 'string', '_default': 'hello world!'},
    'c': 'array[(3|4),U36]'}

map_schema = {
    '_type': 'map',
    '_key': 'string',
    '_value': 'float'}
```

From `tests.py` (`node_schema`, `map_schema`).

A key that starts with `_` is a **schema key** (metadata about the type); any other key is a named child in the place graph. That is the whole rule (`is_schema_key` = "starts with `_`").

3 · Compiled `Node`

An already-compiled dataclass instance — what `access` returns. Passing one back in is a no-op, which is why operations can accept any of the three forms interchangeably:

```
node_type = core.access(node_schema)   # dict  -> Node
same      = core.access(node_type)     # Node  -> Node (idempotent)
```

> **`access` is the great normalizer**
>
> Every `Core` method that takes a schema calls `access` on it first. That is why you can hand `core.default`, `core.check`, `core.serialize`, `core.traverse`, and the rest either a terse string, a verbose dict, or a compiled `Node` and get the same behavior. Write schemas in whichever form is clearest for the reader.

Base types

`BASE_TYPES` seeds the registry with a family of dataclasses, all subclassing `Node`. The exact set drifts as the package grows, so treat this as the shape of the family, not a closed list:

Scalars / atoms

`empty`, `boolean` (and `or` / `and` / `xor`), `integer`, `float` / `float64`, `number`, `complex`, `delta` (a float used for additive updates), `nonnegative`, `range`, `string`, `enum`, `dtype`. `number` carries `_units` (a pint string) and `_bits`, so `integer[64]` and `float[32]` are spellable.

Wrappers

A `Wrap` holds an inner `_value` node and changes its behavior: `maybe` (nullable, default `None`), `overwrite` (apply replaces), `const` (apply is a no-op), `quote` (opaque passthrough), plus divide-behavior wrappers (`divide_reset`, `divide_share`, `lineage_seed`).

Containers

`list`, `set`, `map` (`_key` + `_value`), `tree` (a `_leaf` type nested arbitrarily deep), `array` (numpy `_shape` + `_data` + `_units`), `dataframe`, `tuple`, `union`.

Wiring & runtime types

`path` (a wire address — apply replaces, not concatenates), `wires`, `schema`, and `link` (the edge node with `_inputs` / `_outputs` port types and `inputs` / `outputs` wires). Plus `quantity` (a pint value), `function`, `object`, `random_state`.

The **Milner structural types** — `site` (a hole in the place graph), `inner_name` / `outer_name` (open link-graph endpoints), and `face` (an interface `⟨m, X⟩`) — are what turn a grounded schema into a composable context. They are the formal backbone of the site/fill/ground story from Core concepts.

Type inheritance

Types inherit two ways. **Python class inheritance** among the dataclasses drives method dispatch — `Delta ⊂ Float ⊂ Number ⊂ Atom ⊂ Node`, so a method defined for `Number` covers `Float` unless overridden. **Schema-level `_inherit`** lets a *registered* type extend others: a dict with `{'_inherit': [ancestor, …], …}` resolves each ancestor left-to-right, then its own fields override.

Producing state: `default` and `realize`

A schema describes what is possible; you still need an actual value. Two entry points make one.

`default` — the type's zero

`core.default(schema)` returns `(compiled_schema, state)` where `state` is the type's zero value: `Float→0.0`, `Integer→0`, `String→''`, `Boolean→False`, `Map→{}`, `Tree→` its leaf's zero (`tree[float]→0.0`; an empty tree with no leaf value is `{}`), `List→[]`, `Array→np.zeros(shape, dtype)`, `Enum→` its first value, `Maybe→None`. An explicit `_default` field always wins over the zero.

```
default_schema, default_state = core.default(node_schema)
assert default_state['a'] == 11.111          # the declared _default
assert isinstance(default_state['b'], str)
assert core.check(node_schema, default_state) # the default inhabits its schema
```

From `tests.py::test_default`.

`realize` — generate / fill / complete

`realize` is the workhorse: given a schema and a *partial* state, it decodes existing values to their declared types and fills every missing key with a default. It is the canonical "complete this graph" entry point, and it runs in **two phases** — first a `discover` pass walks the state, coerces present values, and collects the defaults a `link`'s wired ports declare; then the fill pass supplies missing keys using those port-enhanced defaults. That split is why a wire like `'float{5.5}'` can override a bare schema default.

```python
schema = {
    'A': 'float',
    'B': 'enum[one,two,three]',
    'D': 'string{hello}',
    'units': 'map[number]'}

state = {
    'C': {'_type': 'enum[x,y,z]', '_default': 'y'},
    'concentrations': {'glucose': 0.5353533},
    'link': {
        '_type': 'link',
        '_inputs':  {'n': 'float{5.5}', 'x': 'string{what}'},
        '_outputs': {'z': 'string{world}'},
        'inputs':   {'n': ['A'], 'x': ['E']},
        'outputs':  {'z': ['F', 'f', 'ff']}},
    'units': {'meters': 11.1111, 'seconds': 22.833333}}

generated_schema, generated_state, _ = core.realize(schema, state)

assert generated_state['A'] == 5.5              # filled from the wired port
default
assert generated_state['B'] == 'one'            # enum default = first value
assert generated_state['C'] == 'y'              # explicit _default wins
assert generated_state['units']['seconds'] == 22.833333
```

From `tests.py::test_generate` — the single best "everything together" example: bare types, enum/string defaults, an embedded wired `link`, and a `map`, all completed in one call.

`realize` returns a **triple** — `(schema, state, escape_merges)` — where the third element carries any *escaped* port merges: merges whose wires point outside the realized subtree, handed back for the caller to apply at a higher scope. For a top-level `realize` (the usual case) nothing escapes, so it is empty — which is why the examples here bind it to `_`. A related helper, `core.fill(schema, state, overwrite=False)`, builds the default state and merges your partial state over it (or under it, if `overwrite`); `process-bigraph`'s `Edge.__init__` uses it to complete a process's config.

`infer` — schema from an example

Going the other way, `core.infer(state)` derives a schema that a given value inhabits. It round-trips with `default`: `core.default(core.infer(value))[1] == value` (the state element of the `(schema, state)` pair `default` returns).

Serialize / render — the codec, and its round-trip law

There are two directions to encode, and it helps to keep them straight:

- `serialize(schema, state)` encodes a *state value* to something JSON-compatible.
- `render(schema, defaults=False)` encodes a *schema* (a `Node`) to JSON/string — it is the inverse of `access`.

```
link_type  = core.access(link_schema)
encoded    = core.serialize(link_type, link_a)
assert encoded['address']  == 'local:edge'
assert encoded['_inputs']  == 'mass:float|concentrations:map[float]'
```

From `tests.py::test_serialize` — note the port schema serializes back to the same compact DSL string you could have typed.

The important guarantee is the **access ⇄ render inverse law**: normalizing a schema and rendering it back are inverses, so a schema survives a round trip through JSON unchanged.

```
def do_round_trip(core, schema):
    type_      = core.access(schema)               # string/dict -> Node
    reified    = core.render(type_, defaults=True) # Node -> JSON
    round_trip = core.access(reified)              # JSON -> Node
    final      = core.render(round_trip, defaults=True)
    return type_, reified, round_trip, final
```

From `tests.py::do_round_trip`. `test_render` asserts `core.access(link_render) == link_type` and that rendering reaches a fixed point.

The decode direction is `realize`, and it is forgiving: it will parse JSON *strings* found where structured values are expected. Feed a wire as the literal string `'["cell", "internal"]'` and `realize` parses it back into a list; feed `'5555'` where an `integer` is declared and you get `5555`.

```
schema = {'a': 'integer', 'b': 'tuple[float,string,map[integer]]'}
code   = {'a': '5555', 'b': ('1111.1', 'okay', '{"x": 5, "y": "11"}')}
decoded_schema, decoded_state, _ = core.realize(schema, code)
assert decoded_state['a'] == 5555
assert decoded_state['b'][2]['y'] == 11
```

From `tests.py::test_realize`.

> **`bundle` — serialize with big arrays spilled**
>
> For states carrying large numpy arrays, `core.bundle` is `serialize` plus a Parquet spill: the array data lands in a side file and the encoded state references it. Useful when an emit path would otherwise inline megabytes of array into JSON.

The `apply` law and update sentinels

Here is the semantic the whole framework rests on. A Process never mutates state; it returns a typed **delta**, and the runtime folds that delta in through the type's own `apply`. **How a delta combines is a property of the data type, not the process** — which is exactly what lets two independently-written processes write to the same store.

```
def apply(self, schema, state, update, path=(), update_has_structural=None,
events=None):
    ...
```

`core.apply(schema, state, update)` returns `(new_state, merges)`. For a plain scalar the update is just the new value combined by the type's rule (numeric `delta` adds; `overwrite` replaces; `const` ignores). For containers, the update can carry **sentinels** — reserved keys that name a structural operation:

Sentinel	Meaning
`_add`	extend the container with new positions / keys / elements
`_remove`	drop elements; the value `'all'` clears the container first
`_divide`	split one position into daughters (cell division)
`set`	overwrite the whole structure (used by `array`)

```
# Additive dict update — numeric leaves accumulate:
result, _ = apply(schema, state.copy(), {'count': np.array([1, 2, 3])}, ())
assert list(result['count']) == [11, 22, 33]

# 'set' sentinel — replace instead of accumulate:
result, _ = apply(schema, state.copy(), {'set': {'count': np.array([100, 200,
300])}}, ())
assert list(result['count']) == [100, 200, 300]
```

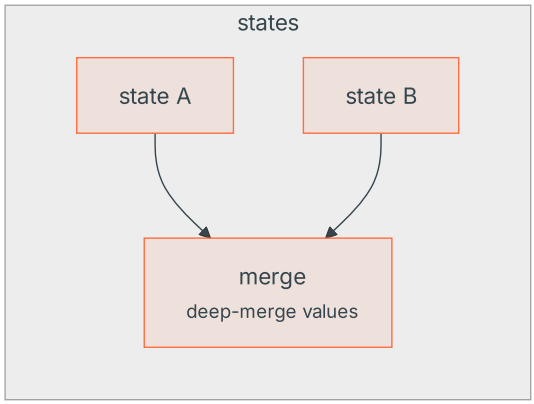
From `tests.py` (array/dict apply). Within one `apply`, sentinels dispatch in a defined order — `_divide` → `_add` → `_remove` — and a `_remove: 'all'` clears the container before adds land.

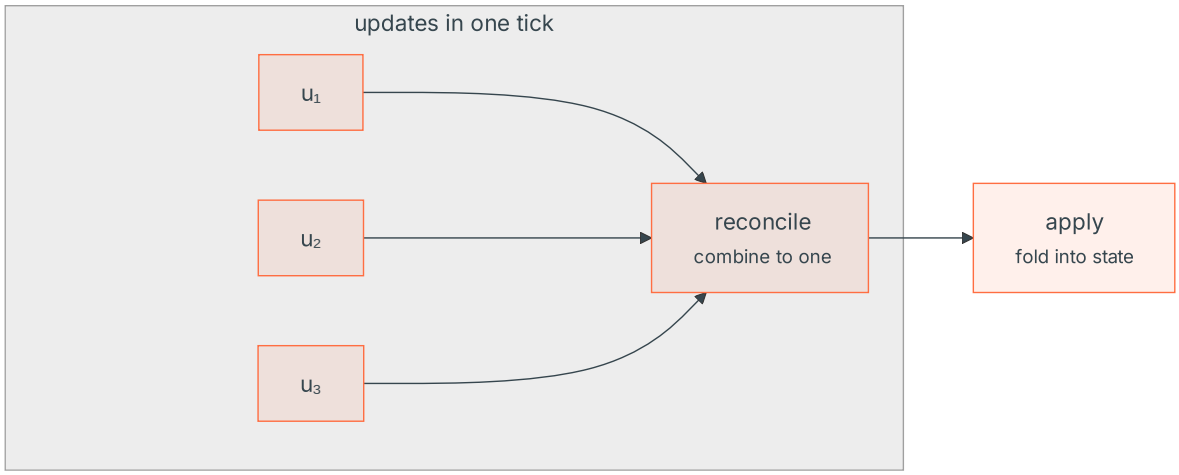
When you pass an optional `events` list, `apply` records the structural changes the sentinels caused — `_add` → `NodeAdded`, `_remove` → `NodeRemoved`, `_divide` → `Divided` — so a Composite can rebuild its process index after the state's shape changes.

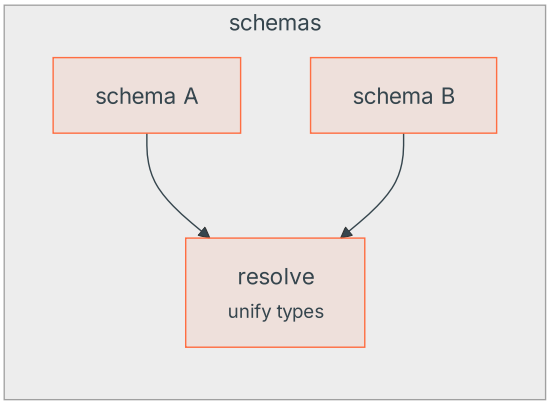
> ⚠ **Accuracy note — `_divide` is real, `_react` is not**
>
> `divide` is a **first-class, shipped method** (`methods/divide.py`), triggered by the `_divide` sentinel in `apply` on `map` / `tree` states — this is how cell division splits a mother store into daughters. A `_react` sentinel (tree-rewrite reaction rules) is **specified in `doc/method_api_spec.md` but not implemented** as a method module in this tree. Treat `_react` as aspirational; do not write code that depends on it.

The composite algebra: `resolve`, `reconcile`, `merge`

Three operations combine typed things. They are easy to confuse because they all "put two things together," so hold the distinction firmly: **`resolve` combines two schemas, `merge` combines two states, and `reconcile` combines a list of updates.**







`resolve` — unify two schemas

`core.resolve(current, update)` unifies two schemas field-wise. `resolve('float', 'number')` is `float` (the more specific wins); resolving two structured dicts unions their keys; genuinely conflicting types raise.

```
float_number = core.resolve('float', 'number')
assert render(float_number) == 'float'

mutual = core.resolve({'a': 'float', 'b': 'string'},
                      {'b': 'wrap[string]', 'c': 'boolean'})
assert {'a', 'b', 'c'} <= set(mutual)          # keys unioned

# conflicting types (map[string] vs float) -> raises
```

From `tests.py::test_resolve`. `resolve` is heavily memoized because the engine calls it on every tick.

A sibling, `core.promote(library, sparse)`, is a per-tick optimization: it walks only the *sparse* schema's keys and substitutes the richly-typed nodes from a library where they exist — so a per-cell wire projection like `{'fields': {'glucose': {0: {0: 'float'}}}}` gets the library's typed `Map` node at `fields`, and nothing the sparse update never touched is pulled in.

`reconcile` — combine a list of updates into one

Within one tick, several processes can emit updates to the same store. `core.reconcile` combines that list into a single update *before* one `apply`. Numeric deltas **sum** (returning `None` when they cancel to zero); overwrites take the last non-`None`; maps merge per-key; lists union their adds.

```
reconcile(Float(), [1.0, 2.5, -0.5]) == 3.0      # deltas sum
reconcile(Float(), [None, None]) is None          # cancel -> None
reconcile(Map(), [{'a': 1}, {'b': 2}, {'a': 3}])  # -> {'a': 3, 'b': 2}
```

From `tests.py::test_reconcile_*`.

The governing law is that reconcile must agree with folding `apply` over the list:

`apply(x, reconcile(s, [u₁, u₂])) == apply(apply(x, u₁), u₂)`

For **commutative** types (numerics, sets, symmetric add/remove) this holds regardless of order. For **non-commutative** types (strings, overwrites, asymmetric structural sentinels) reconcile is instead the correct *batched* semantics — "these all happened concurrently in this tick" — which a plain fold cannot express.

`merge` — deep-merge two states

`core.merge(schema, state, merge_state)` is the state analogue of `resolve`: a schema-aware deep merge of two *values*. And `core.combine(schema, state, update_schema, update_state)` does all of it at once — resolve the schemas, merge the states, realize the result — which is what you reach for when two typed subgraphs need to become one.

The two design docs — the authoritative theory

Two documents in the repo are the ground truth for this algebra, and they are worth reading if you extend a container type:

- `doc/composite_algebra.md` takes the categorical view: each type is a triple `(X, U, apply)`; updates form a **monoid** under `reconcile` with identity `ε` (`None` / `0`); schema constructors like `list[·]` and `map[·]` lift these monoids **functorially**; a composite is a **coalgebra** whose tick is `read → run → reconcile → apply`. It ends with a 12-item punch list of unifications the structure suggests but that are not yet built (top item: a uniform `StructuralUpdate[F, T]` mixin so every container shares one add/remove/set algebra).
- `doc/reconcile_audit.md` is the bug-driven view: a per-type audit of `reconcile` for batch correctness (CRASH / DROP / AMBIGUOUS / OK). Its header status (2026-05-08) records that all CRASH/DROP findings — in List, Tree, Set, Tuple, Array, and dict-schema — are **fixed**, and Map gained `_remove: 'all'`. What remains are documented soft-contention flags: concurrent scalar writes resolve last-non-`None`-wins by process order. Read it for edge-case caveats before relying on concurrent writes to the same store.

Navigating state: `jump` and `traverse`

State is a nested dict tree — the place graph — and you address a value by its **path**, a tuple of keys. Two schema-aware navigators walk it and return *both* the sub-schema and the sub-state, so you never lose the type on the way down:

- `core.jump(schema, state, key)` takes one step.
- `core.traverse(schema, state, path)` walks a full path, and understands `..` (climb) and `*` (wildcard fan-out).

```python
tree_a = {'a': {'b': 5.5, 'x': {'further': {'down': 111111.111}}}, 'c': 3.3}

# descend a tree[float]:
down_schema, down_state = core.traverse('tree[float]', tree_a,
                                        ['a', 'x', 'further', 'down'])
assert isinstance(down_schema, Float) and down_state == 111111.111

# wildcard across a map's values, then into a field:
star_schema, star_state = core.traverse(
    {'_type': 'map', '_value': {'a': 'float', 'b': 'string'}},
    {'X': {'a': 5.5, 'b': 'green'}, 'Y': {'a': 11.11, 'b': 'g2'}},
    ['*', 'a'])
assert star_state['Y'] == 11.11

# jump into a link node, then traverse its wired inputs:
down_schema, down_state = core.jump(simple_interface, simple_graph, 'link')
assert isinstance(down_schema, Link)
mass_schema, mass_state = core.traverse(simple_interface, simple_graph,
                                        ['link', 'inputs', 'mass'])
assert isinstance(mass_schema, Float)
```

Condensed from `tests.py::test_traverse`.

The path helpers underneath — `get_path`, `establish_path` (creates, honors `..`), `set_path`, `resolve_path` (canonicalizes `..`) — are exported from `bigraph_schema` if you need to manipulate paths directly.

Views and projections: ports as lenses

A Process does not see the whole state — it sees the slice its input wires point at, and its output update is written back through its output wires. `Core` implements both directions as a lens:

- `core.view(schema, state, link_path, ports_key='inputs')` reads the slice a link's wires select — what a process receives.
- `core.project(schema, state, link_path, view, ports_key='outputs')` inverts a port-local update back into a composite-wide update — how a process's output lands in shared state.

```
basic_link = {
    '_type': 'link',
    '_inputs':  {'x': 'float', 'y': 'array[(6),float]'},
    '_outputs': {'z': 'float', 'w': 'array[(5),float]'},
    'inputs':   {'x': ['array', 4, 3], 'y': ['array', 2]},
    'outputs':  {'z': ['array', 1, 5], 'w': ['array', '*', 3]}}

basic_schema, basic_state, _ = core.realize(
    {'array': 'array[(5|6),float]'},
    {'array': np.zeros((5, 6)), 'link': basic_link})

view        = core.view(basic_schema, basic_state, ('link',))
# inputs
output_view = core.view(basic_schema, basic_state, ('link',),
ports_key='outputs')

project_schema, project_state = core.project(
    basic_schema, basic_state, ('link',),
    {'z': 5555.5, 'w': np.array([1., 2., 3., 4., 5.])})

applied_state, applied_merges = core.apply(project_schema, basic_state,
project_state)
```

Condensed from `tests.py::test_array` — view reads through the input wires, project turns a `{z, w}` output into a whole-state update, and apply folds it in.

These are the lens laws in practice: a no-op port write yields a no-op composite update, and reading after writing returns what was written. `Core` also compiles cached fast paths (`view_fast`, `project_ports_fast`) that fold unit conversions in at compile time — that is how a wire between a store in `mM` and a port in `mol/L` converts without either side knowing about the other.

Where this connects

Everything above is machinery `process-bigraph` drives for you. When you wire a Composite, the engine `view`s each process's inputs, calls its `update`, `reconcile`s the tick's updates, and `apply`s the result — the loop from `doc/composite_algebra.md`. You rarely call `apply` or `reconcile` by hand; you rely on having *declared the right types* so those operations do the right thing automatically.

→ Build dynamics

Processes & Steps — write an `Edge` that declares typed `inputs()` / `outputs()` and returns deltas this layer merges.

→ Wire them up

Composites & wiring — connect ports to store paths and nest composites, where `view` / `project` and `resolve` do their work per tick.

> **The one sentence to keep**
>
> A schema says what state can be; `apply` says how changes to it combine — and because combination is a property of the *type*, not the *process*, independently-written simulators compose without coordination.

Next: Processes & Steps

process step

Processes & Steps

A model in Vivarium is made of two things: **state** that lives in stores, and **functionality** that acts on it. The functionality is packaged as **edges** — and there are exactly two kinds, split by their relationship to time. This chapter shows you how to write both, from the base-class contract down to real, runnable code.

On this page

Assumes Core concepts and Schemas, types & state.

You'll learn:

- What an **edge** is, and why it never talks to another edge directly
- The `Process` contract — `config_schema`, `inputs()`, `outputs()`, `update(state, interval)`, `initial_state()`
- The `@process` decorator shortcut that infers `config_schema` for you
- How a **Step** differs from a Process, and how Steps form a dataflow DAG that runs to quiescence
- Registering an edge with `core.register_link(name, cls)`

What an edge is

In Core concepts an **edge** is defined as a unit of functionality attached to stores. Each of its inputs and outputs is a **port**. The rule that makes the whole framework compose is that **an edge never talks to another edge directly** — it reads and writes **shared stores**, and that shared wiring *is* the coupling between edges.

That gives an edge a very small, very strict job:

Read

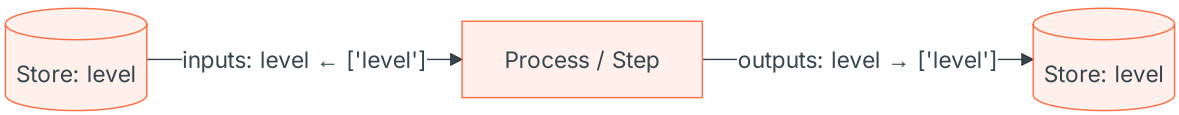
The runtime hands `update()` a `state` dict whose keys are the edge's **input port names**, with values pulled from the stores those ports are wired to.

Compute

The edge does its work — an integration step, an FBA solve, a unit conversion, a report card — using only what it was given plus its own `config`.

Return a delta

It **returns** a dict keyed by **output port names**. It does *not* write anything itself. The runtime merges that delta into shared state through the type's `apply` rule.



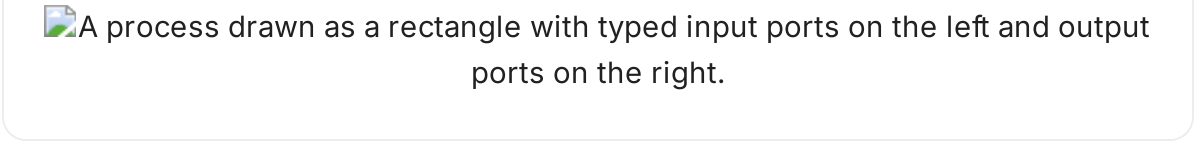
> *Because an edge only ever returns a delta, two independently-written edges can write to the same store without knowing about each other. That is the one semantic that lets the whole framework compose.*

The two kinds of edge differ in exactly one thing — **time**:

	Process — *temporal*	**Step — *reactive***
Declares an `interval`?	yes, a timestep	no
`update` signature	`update(self, state, interval)`	`update(self, state)`
When it runs	on its own interval schedule, driven by the clock	when its input wires change — a dataflow rule
Good for	ODE integrators, FBA steps, stochastic steps, agent dynamics	reactions, unit conversions, analyses, report cards, emitters

The `Process` base class

A **Process** is a temporal edge: it owns a timestep and advances dynamics over a span of time. Conceptually it is *a mechanism with an interface* — defined by what it reads, what it can update, and its configuration — that observes part of the current state, uses its config to decide how to respond, and proposes changes through its outputs. You write one by subclassing `process_bigraph.Process` and filling in a handful of methods. Here is the canonical teaching process shipped in the repo (its one `accelerate` helper trimmed, so only the contract shows):



Process diagram. *A process is a rectangle with typed* ***ports*** *on its boundary — inputs on the left, outputs on the right. Its update function maps inputs to a typed delta of outputs, informed by its config (here, an* `interval`*).* *(Agmon & Spangler, Fig 4.)*

```python
from bigraph_schema import make_default
from process_bigraph.composite import Process, Step


class IncreaseProcess(Process):
    config_schema = {
        'rate': {
            '_type': 'float',
            '_default': '0.1'}}

    def inputs(self):
        return {
            'level': 'float'}

    def outputs(self):
        return {
            'level': 'float'}

    def initial_state(self):
        return {
            'level': 4.4}

    def update(self, state, interval):
        return {
            'level': state['level'] * self.config['rate']}
```

Source: `process_bigraph/processes/examples.py`.

Each piece of that class is one clause of the Process contract:

`config_schema` — a class attribute

`config_schema` is a **class attribute** (not a method) holding a bigraph-schema schema dict that describes the process's configuration. Values are validated and defaulted against it, and the instance reads them back at `self.config[...]`:

```
config_schema = {'rate': {'_type': 'float', '_default': '0.1'}}
#   → self.config['rate'] is available inside update()
```

A bare type name is a valid shorthand — `config_schema = {'rate': 'float'}` — and `bigraph_schema.make_default('float', 0.001)` builds a defaulted field for you.

`inputs(self)` and `outputs(self)`

Two methods that return the **port interface** — a dict mapping each port name to a type expression. Ports specify the type of information flowing in and out, so that data is passed consistently between edges. Inputs declare what the process reads; outputs declare what it may write:

```
def inputs(self):  return {'level': 'float'}
def outputs(self): return {'level': 'float'}
```

The keys here are the same names that appear in the `state` dict passed to `update()` and in the delta it returns. Port types can be as rich as the type system allows — `'map[float]'`, `'array'`, `'map[mass:float]'`, or a dict carrying `_units`.

`update(self, state, interval)` — the temporal step

This is where the dynamics live. `state` is a dict keyed by **input port names**, `interval` is the timestep, and the return value is a **delta dict keyed by output port names**:

```
def update(self, state, interval):
    return {'level': state['level'] * self.config['rate']}
```

The returned dict is a *change*, not a new value — the runtime merges it into the wired store through that type's `apply` rule (numeric types accumulate, `set` replaces, dicts merge). See the merge law in Core concepts.

`initial_state(self)` — optional

If present, `initial_state()` returns the starting values this process contributes to the stores it is wired to. When a composite is built, each edge's `initial_state()` is combined into the global state, so a process can seed its own inputs:

```
def initial_state(self):
    return {'level': 4.4}
```

One more optional hook

`initialize(self, config)` lets a process precompute expensive state from its config (a loaded scientific model, a JIT cache, a solver base). The default `reconfigure()` re-runs `initialize`, which is what lets a pooled actor be re-purposed per-sim without a cold start. Most processes never need either — `config_schema` is enough.

The `@process` decorator shortcut

When a process is a **pure typed function** of `(state, interval)` plus some configuration, you don't need the class boilerplate at all. The `@process` decorator turns a function into a `Process` subclass and **infers `config_schema` from the function's keyword-only arguments**:

```
from process_bigraph import allocate_core, Composite, process


@process(inputs={"S": "float"}, outputs={"S": "float"})
def decay(state, interval, *, rate: float = 0.1):
    return {"S": -rate * state["S"] * interval}   # additive delta
```

Source: `notebooks/tutorial_0_quickstart.ipynb`.

The keyword-only parameters after `*` **are** the config: `rate: float = 0.1` becomes a `float` field defaulting to `0.1`. The inferred schema is available on the class:

```
decay.config_schema     # → the inferred {'rate': {'_type': 'float', '_default': 0.1}}
```

Python-scalar annotations map onto bigraph types (`float → 'float'`, `int → 'integer'`, `bool → 'boolean'`, `str → 'string'`); a *string* annotation is taken verbatim

as a type name, which is the escape hatch for richer types like `"map[float]"`. The class-based `Process` remains the escape hatch for anything stateful.

> ⚠ **Accuracy note — register with `register_link`, not `register_process`**
>
> The `@process` docstring says to register the class with `core.register_process(name, cls)`. **That method does not exist.** The real registrar on the `Core` is `core.register_link(name, cls)`, and every working call site — the README, the tests, the emitter docs — uses `register_link`. Treat `register_process` as a docstring typo.

`Step` vs `Process`

A **Step** is the reactive edge. Its docstring calls it *"a stateless, non-temporal computational unit … triggered when its data dependencies are satisfied, functioning like a reaction or transformation rule."* The single visible difference is that `Step.update` **takes no** `interval`:

```
class OperatorStep(Step):
    config_schema = {'operator': 'string'}

    def inputs(self):
        return {'a': 'float', 'b': 'float'}

    def outputs(self):
        return {'c': 'float'}

    def update(self, inputs):          # note: no interval
        a = inputs['a']
        b = inputs['b']
        if self.config['operator'] == '+':
            c = a + b
        elif self.config['operator'] == '*':
            c = a * b
        elif self.config['operator'] == '-':
            c = a - b
        return {'c': c}
```

Source: `process_bigraph/processes/examples.py`.

Steps form a dataflow DAG that runs to quiescence

A Process runs on the clock. A Step runs when **its inputs change**. Because a Step's inputs and outputs are wired to shared stores, the composite can read off which steps depend on which — a step that writes store `y` comes before a step that reads `y` — and build a **dependency DAG**. When the state a step depends on changes, the network fires in topological order and **settles to quiescence** before time advances.

The clearest proof of this is a test in the repo: three steps chained `a → b → c`, where `step_a` writes `y`, `step_b` reads `y` and writes `z`, and `step_c` reads `z`:

```
class ProducerStep(Step):
    config_schema = {'name': 'string'}
    def inputs(self):  return {'in_val': 'float'}
    def outputs(self): return {'out_val': 'float'}
    def update(self, state):
        execution_log.append(self.config['name'])
        return {'out_val': state.get('in_val', 0.0) + 1.0}

core = allocate_core()
core.register_link('ProducerStep', ProducerStep)

composite = Composite({'state': {
    'x': 1.0, 'y': 0.0, 'z': 0.0,
    'step_a': {'_type': 'step', 'address': 'local:ProducerStep', 'config':
{'name': 'a'},
               'inputs': {'in_val': ['x']}, 'outputs': {'out_val': ['y']}},
    'step_b': {'_type': 'step', 'address': 'local:ProducerStep', 'config':
{'name': 'b'},
               'inputs': {'in_val': ['y']}, 'outputs': {'out_val': ['z']}},
    'step_c': {'_type': 'step', 'address': 'local:ProducerStep', 'config':
{'name': 'c'},
               'inputs': {'in_val': ['z']}, 'outputs': {'out_val': ['w']}},
}}, core=core)

composite.run(0.0)                    # a pure-step network settles at interval
0

assert execution_log == ['a', 'b', 'c']
assert composite.state['w'] == 4.0     # a:1→2, b:2→3, c:3→4
```

Source: `tests.py::test_dependency_cycle`.

Two things to notice. First, the wiring alone determines the order — nobody wrote a schedule. Second, a network made only of steps runs to completion at `run(0.0)`: with no process to advance time, "run" just means "settle the DAG." This is exactly

why **emitters, analyses, visualizations, and report cards are all Steps** — each should fire whenever the state it observes is ready, not on a clock of its own.

> **By default, every input triggers**
>
> A Step re-fires when *any* of its input wires updates. Override `triggers()` to return only a subset of ports if some inputs should be *received but silent* (read without causing a re-trigger). There is also a `@step` decorator — the exact analogue of `@process`, with an `update(state)` that takes no interval.

Registering an edge

Whether class-based or decorated, an edge becomes usable by **registering it into a `Core`** under the name that its `address` will reference:

```
from process_bigraph import allocate_core

core = allocate_core()                  # a fresh, isolated type + link registry
core.register_link('Grow', Grow)        # now 'local:Grow' resolves to this class
core.register_link('decay', decay)      # decorated processes register the same
way
```

`allocate_core()` returns a fresh, **isolated** `Core` — types and links registered on one core never leak to another, so two composites in the same process can't collide. Discovery is lazy: a registered link's module is only imported when an `address` first resolves to it. In a document, an edge names its class through an `address` like `local:Grow` — `local:` means "resolve this name in this core's registry."

> **Accuracy note — the registry is on bigraph-schema's `Core`**
>
> Composites are built on a `Core` produced by `allocate_core()` (from `bigraph_schema`), not on a legacy `TypeSystem` object. `core.register_link(name, cls)` is the one and only method for registering a process or step class. See Schemas, types & state for what else the `Core` holds.

Putting it together

A process or step on its own is inert — it needs stores to read and write, and a composite to schedule it. Here is the full loop, using a class-based `Grow` process wired over a single shared store, with an emitter recording the trajectory:

```
from process_bigraph import Composite, Process, allocate_core
from process_bigraph.emitter import emitter_from_wires, gather_emitter_results


class Grow(Process):                                    # a Process = ports + an
update
    config_schema = {'rate': 'float'}
    def inputs(self):  return {'level': 'float'}
    def outputs(self): return {'level': 'float'}
    def update(self, state, interval):
        return {'level': state['level'] * self.config['rate'] * interval}  # a
delta


core = allocate_core()
core.register_link('Grow', Grow)                        # register it in the local
registry

composite = Composite({'state': {
    'level': 1.0,                                       # a shared store
    'grow': {'_type': 'process', 'address': 'local:Grow', 'config': {'rate':
0.5},
             'interval': 1.0, 'inputs': {'level': ['level']}, 'outputs':
{'level': ['level']}},
    'emitter': emitter_from_wires({'level': ['level'], 'time':
['global_time']}),
}}, core=core)
composite.run(5.0)
print(gather_emitter_results(composite))
```

Source: `README.md` quickstart.

That `{'state': {...}}` document — the stores, the `process` / `step` nodes, and the wiring between them — is the subject of the next chapter. How the emitter records state, and how you get the trajectory back out, is covered in Emitters.

Next: Composites & wiring

composite process step wiring

Composites & wiring

A process or step is just a rule. To *run* one you need stores for it to read and write, other edges to couple it to, and a scheduler to advance time. All of that is a **Composite** — the state-tree of typed nodes wired to shared stores that is **the only object the engine actually runs**. This chapter is about the shape of that document, how ports wire to store paths, how composites nest, and what happens on every tick when you call `run`.

On this page

Assumes Processes & Steps.

You'll learn:

- The composite **document shape** — a `state` tree of `process` / `step` nodes
- Wiring ports to shared-store **paths**, including relative paths like `['..']`
- **Nesting** composites through the `bridge`
- Running with `composite.run(interval)`
- The **tick lifecycle** — the invoke pass, the apply pass, and how steps get triggered

The composite document

A composite is a **document**: a plain Python dict (JSON-serializable) whose main key is `state`. Inside `state`, some entries are plain **stores** — a typed value like a count or a concentration — and some are **edge nodes**, each a `process` or `step` with an address, config, and wiring:

```
composite = Composite({'state': {
    'level': 1.0,                                   # a store — a plain typed
value
    'grow': {                                       # an edge node
        '_type':    'process',                      # 'process' or 'step'
        'address':  'local:Grow',                   # protocol:name, resolved
in the core
        'config':   {'rate': 0.5},                  # validated against the
class's config_schema
        'interval': 1.0,                            # a process's timestep
(steps omit this)
        'inputs':   {'level': ['level']},           # port → path into a shared
store
        'outputs':  {'level': ['level']},           # port → path into a shared
store
    },
}}, core=core)
```

Each field of an edge node has one job:

Field	Meaning
`_type`	`'process'` (temporal, clocked) or `'step'` (reactive, dataflow).
`address`	`protocol:name`. `local:` resolves `name` in this core's link registry — so `local:Grow` needs `core.register_link('Grow', Grow)`.
`config`	Values validated and defaulted against the class's `config_schema`; read back as `self.config[...]`.
`interval`	A process's timestep. **Steps have no `interval`** — they fire on data change, not the clock.
`inputs`	A dict `{port_name: path}` mapping each input port to the store path it reads.
`outputs`	A dict `{port_name: path}` mapping each output port to the store path its delta is merged into.

A **store** in the document is just a plain typed entry in `state` — `'level': 1.0`, or a richer `{'_type': 'float', '_units': 'pg'}`. There is no special "store" keyword;

anything that isn't an edge node is state.

> **Static vs generated documents**
>
> The document above is a *static* composite — you write the `state` dict out by hand or load it from a `.composite.json` file. The workbench also builds documents *programmatically* (a generator or `CompositeSpec` that assembles `state` from parameters), and *templates* leave holes — **sites** — to be filled later. Same document, filled two different ways; see Templates & draft processes.

Wiring: ports to store paths

The coupling rule from Core concepts is worth restating exactly, because it is the whole model: **edges never talk to each other directly.** They read and write shared stores, and *that shared wiring is the coupling.*

A **wire** is a **path** — a list of strings naming a location in the state tree:

- `['level']` targets `state['level']`.
- `['Env', 'x']` targets `state['Env']['x']` — the path descends into nested stores.

An edge node's `inputs` and `outputs` map each **port name** to such a path:

```
'inputs':  {'level': ['level']},     # the 'level' port reads  state['level']
'outputs': {'level': ['level']},     # the 'level' port writes state['level']
```

Change a path and you rewire the model; point two edges' ports at the **same** path and they are now coupled — one process's delta lands in the store the other reads, with no direct reference between them. Nothing else connects edges.

> *Wiring is the coupling. To connect two processes you don't call one from the other — you point their ports at the same store path.*

Relative paths climb and descend

Paths are relative to the node that declares them, so a nested edge can reach **out** to a parent store with `'..'` (one per level up) or **in** to a child store by naming it. The repo's nested-compartment test wires an agent to both its parent environment and its own inner compartment:

```
'inputs': {
    'outer': ['..', '..'],           # climb two levels to the enclosing
environment
    'inner': ['inner']},             # descend into this node's own inner store
'outputs': {
    'outer': ['..', '..'],
    'inner': ['inner']},
```

Source: `tests.py::test_reaction` (`SimpleCompartment`).

This is what makes **containment** work: an agent nested inside an environment inside another agent can still read the field it lives in, without anyone flattening the hierarchy. Wires may also contain `'*'`, matched across every key of a `map`-typed store — the pattern used to wire a dynamic population of agents.

Nesting: the bridge

Here is the structural payoff of the whole design: **a `Composite` is itself a `Process`.** It owns an internal state-tree and scheduler, and it exposes external ports through a **bridge** that maps those ports onto internal store paths. Put another way, a composite **packages an internal process bigraph as a single higher-level process** — so a whole simulation drops into a parent composite as a single node, wired exactly like any other edge.

Composite diagrams: a process graph wiring processes to stores, and a composite process exposing external ports through a bridge.

Composite diagrams. *(a) A process graph wires processes (metabolism, gene expression) to stores (DNA, enzymes, nutrients, products) through ports whose type must match the store they connect to. (b) A* ***composite process*** *wraps an internal process bigraph and exposes external ports; matching internal ports link to the inner bigraph by dotted wires — the* ***bridge*** *— keeping inner and outer state synchronized.*

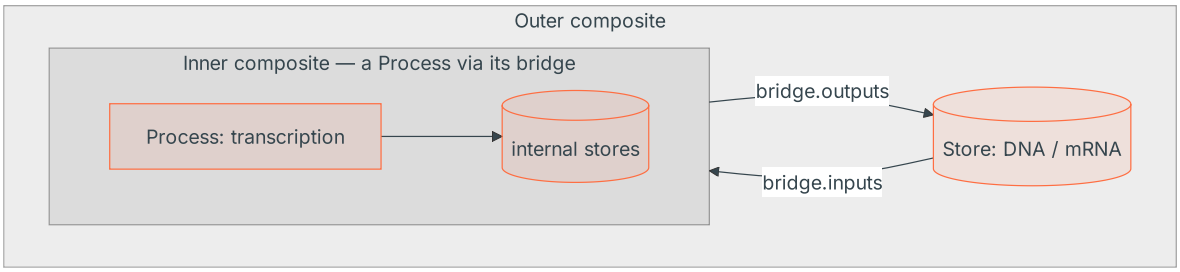
(Agmon & Spangler, Fig 5.)

The bridge is part of the composite's config — two wire-maps, one for each direction:

```
'bridge': {
    'inputs':  {'DNA': ['DNA'], 'mRNA': ['mRNA']},    # external port → internal store path
    'outputs': {'DNA': ['DNA'], 'mRNA': ['mRNA']},
}
```

Source: `tests.py::test_emitter` (a bridged Gillespie composite).

`Composite.inputs()` and `outputs()` return the schema implied by the bridge wiring, so from the outside the composite *is* a process with those ports. When it's used as a node inside a bigger composite, its `update(state, interval)` projects the outer state onto its bridge inputs, runs the interval internally, and returns its bridge outputs as a delta.



> *Multiscale models are built by containment, not by editing a monolith: a cell composite drops into a colony composite the same way a process drops into a cell.*

This is the mechanism behind the framework's central slogan — *composition is closed*. Because a composite is a process, a composite of composites is still a process, all the way up. See Core concepts.

Running a composite

You advance a composite by calling `run` with a span of time:

```
composite.run(5.0)          # advance the simulation by 5 time units
```

`run(interval)` drives the tick loop until the requested time has elapsed. Two details worth knowing:

- **A pure-Step network runs at `run(0.0)`.** With no process to advance the clock, "running" just means settling the step DAG to quiescence — analyses and report cards over a fixed state fall in this case.
- **`run` is resumable.** Each call advances from the current `global_time`; calling it again continues where the last one stopped.

After a run, the composite's `state` holds the final values, `read_bridge()` views the external output ports, and `timing_summary()` reports the split between time spent inside processes and time spent in the framework itself.

A complete, runnable example

Everything above, end to end — a `Grow` process over one shared store, an emitter recording the trajectory each tick, and the results gathered back at the end:

```
from process_bigraph import Composite, Process, allocate_core
from process_bigraph.emitter import emitter_from_wires, gather_emitter_results


class Grow(Process):
    config_schema = {'rate': 'float'}
    def inputs(self):  return {'level': 'float'}
    def outputs(self): return {'level': 'float'}
    def update(self, state, interval):
        return {'level': state['level'] * self.config['rate'] * interval}


core = allocate_core()
core.register_link('Grow', Grow)

composite = Composite({'state': {
    'level': 1.0,                                    # a shared store
    'grow': {'_type': 'process', 'address': 'local:Grow', 'config': {'rate':
0.5},
             'interval': 1.0, 'inputs': {'level': ['level']}, 'outputs':
{'level': ['level']}},
    'emitter': emitter_from_wires({'level': ['level'], 'time':
['global_time']}),
}}, core=core)
composite.run(5.0)
print(gather_emitter_results(composite))
# {('emitter',): [{'level': 1.0, 'time': 0.0}, {'level': 1.5, ...}, ...
{'level': 7.59, 'time': 5.0}]}
```

Source: `README.md` quickstart.

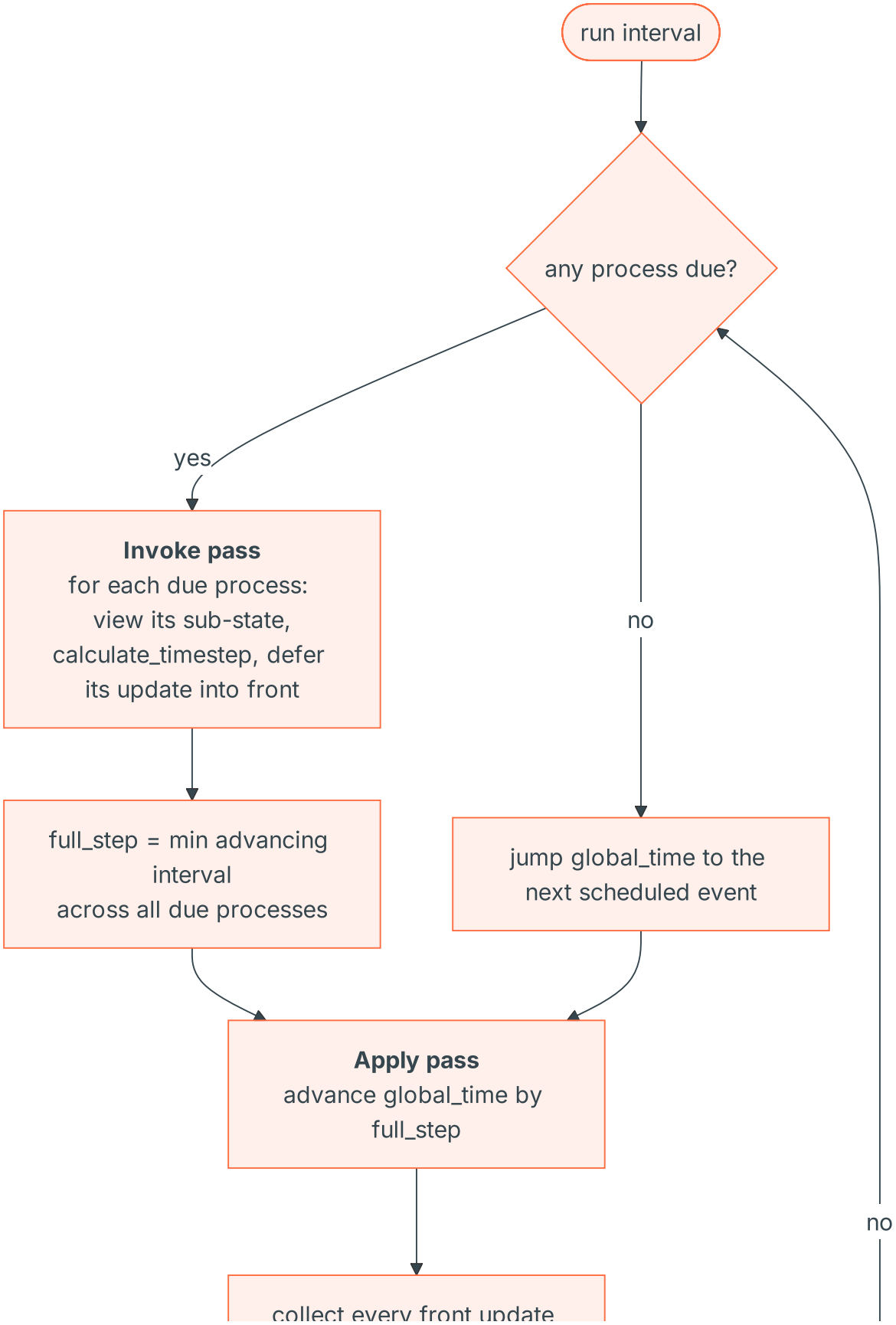
The `emitter` node is an ordinary `step` produced by the `emitter_from_wires` helper; it observes the wired paths every tick and writes them to a sink. That whole story — built-in emitters, how to retrieve results, how to write your own — is the next chapter.

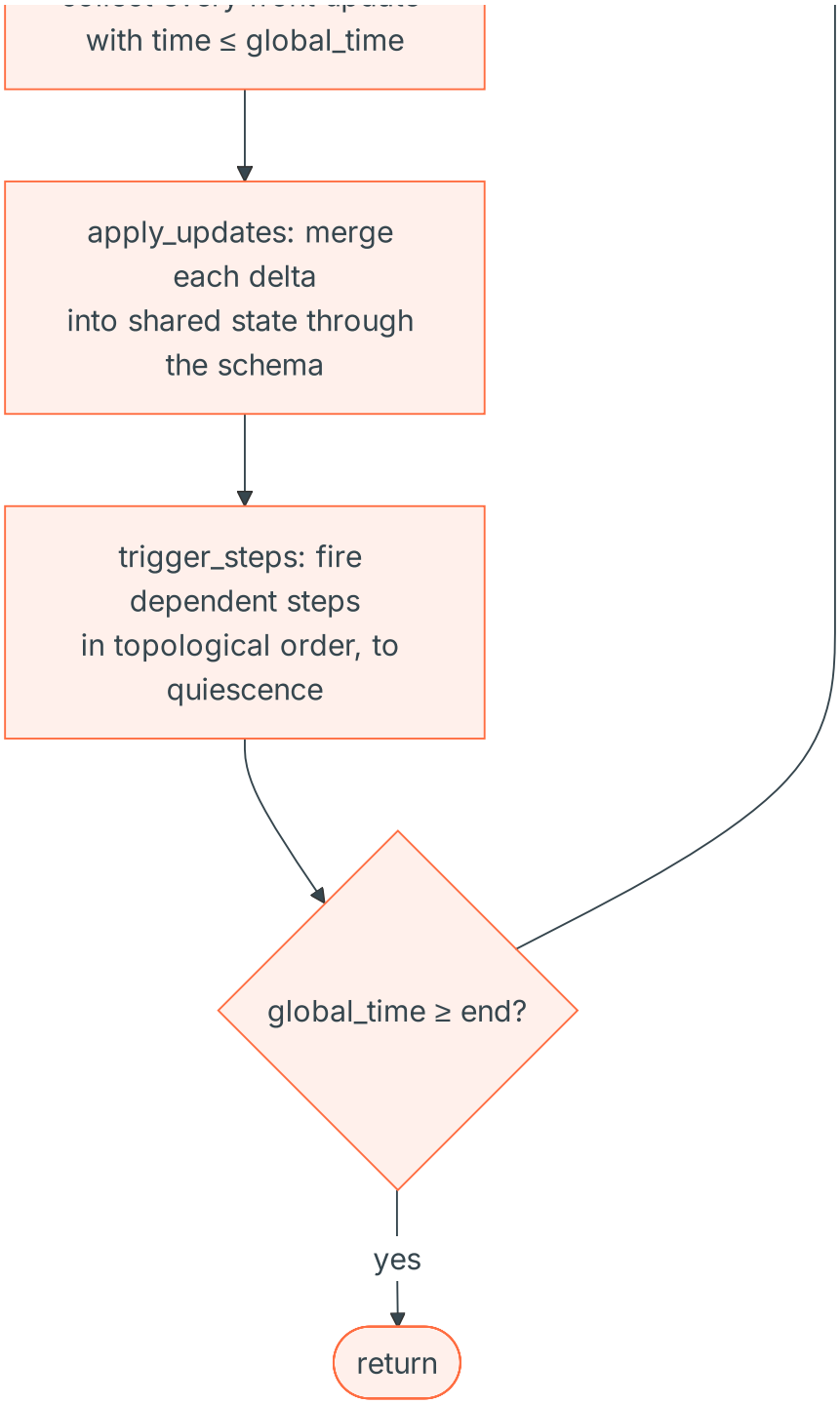
The tick lifecycle

Under `run`, time advances one **tick** at a time, and each tick is **two passes**. This is the heart of the scheduler, and it's worth understanding because it explains why deltas never collide and why steps fire exactly when they should. Where the wiring above describes how processes are *connected*, the tick loop is **orchestration** — how they are *run in time*: at each step the composite decides which processes are eligible, gathers what they need from the state, invokes them, and merges their proposed changes back into the shared stores.


Three orchestration patterns: multi-timestepping, a workflow DAG of steps, and event-driven graph rewrite (divide, engulf, burst).

Orchestration patterns. *(a)* ***Multi-timestepping*** *— temporal processes each update at their own interval, coordinated by a discrete-event co-simulation engine. (b) A* ***workflow*** *is a DAG that orders step processes, each triggered by changes to its inputs. (c)* ***Event-driven graph rewrite*** *— discrete events change the topology of an agent–environment system: divide splits one agent into two, engulf nests one inside another, burst dissolves an agent back into its environment.* *(Fig 6.)*


run interval
any process due?
yes
no
Invoke pass
for each due process:
view its sub-state,
calculate_timestep, defer
its update into front
full_step = min advancing
interval
across all due processes
jump global_time to the
next scheduled event
Apply pass
advance global_time by
full_step
collect every front update
no



1 · Invoke pass. For each process that is due, the composite *views* the sub-state that process is wired to, asks it for its timestep (`calculate_timestep`), and stores a **deferred** update in `self.front` — a per-process timeline dict, `{path: {'time': next_due, 'update': ...}}` — tagged with the future time at which it applies. Crucially, nothing is written to shared state yet; the process has only *returned a*

delta. The pass also accumulates `full_step`, the **minimum** advancing interval across all due processes, so heterogeneous timesteps collapse into one shared advance.

2 · Apply pass. The composite advances `global_time` by `full_step`, collects every `front` update whose scheduled time has now arrived (`time ≤ global_time`), and calls `apply_updates`, which walks the schema and **merges each delta** into shared state through its type's `apply` rule. Only now does state change. Then `trigger_steps` fires every step whose input wires were just updated, in topological order, until the step DAG settles.

If no process is due, time simply jumps to the next scheduled event rather than crawling. The loop repeats until `global_time` reaches the requested end.

Why two passes

Separating *invoke* (collect deltas) from *apply* (merge them) is what lets independently-written processes write to the same store in one tick without racing: every delta for a tick is gathered first, then merged together through the type's `apply` rule. It is the runtime-level expression of the one merge law.

Scaling the loop — protocol batching

When many processes share one distributed runtime (for example, a grid of shards routed through a single Ray runtime), the per-process view-and-apply work can dominate the actual compute. A runtime can implement a `tick_lifecycle()` hook to take over the whole invoke-and-apply lifecycle for its processes and return **one** combined delta, collapsing N framework walks into one. Plain processes and runtimes without the hook keep the per-process path unchanged. Details in the repo's `docs/tick_lifecycle.md`.

Where this sits

You now have the full computational spine: types and state define what stores can hold, processes and steps act on them, and a composite wires them together and runs them. What's left is getting the numbers *out* of a run — which is a job for a particular kind of step.

Next: Emitters — getting data out

composite emitter provenance

Emitters — getting data out

A composite run produces a stream of state that never touches the disk on its own. The engine advances stores, merges deltas, and moves on — nothing is kept unless something asks to keep it. That something is an **emitter**: the Step that watches wired state and writes it to a durable sink. When Core concepts says a run "emits a run store of trajectories," the emitter is the object doing the emitting.

On this page

Assumes Composites & wiring. · **You'll learn** the emitter contract, how an emitter attaches as a plain Step, how it writes wired state to a durable sink, and why it never writes back into the run.

An emitter records wired state each tick into a durable sink — and it is the only part of a composite that is allowed to leave the run.

Because an emitter is a plain Step, it inherits everything a Step already is: it declares input ports, it is scheduled on the data-flow DAG, and it fires when the state it depends on changes. What makes it an *emitter* is a two-line contract and one discipline — it writes to a sink and it **never writes back into the simulation**.

The emitter contract

Every emitter subclasses `process_bigraph.emitter.Emitter`, which is itself a `Step`. The whole base class is small enough to read in one sitting:

```python
class Emitter(Step):
    '''Base emitter class: defines schema and stub methods.'''
    config_schema = {'emit': 'schema'}

    def inputs(self) -> Dict:
        return self.config['emit']

    def outputs(self) -> Dict:
        return {'results': 'node'}

    def query(self, paths=None, query=None):
        return {}      # subclasses return the recorded history

    def update(self, state) -> Dict:
        return {}      # subclasses record; the base no-op keeps it read-only
```

Four things are load-bearing here, and every built-in and custom emitter honours them:

Element	What it means
`config_schema = {'emit': 'schema'}`	The emitter is configured by an **emit schema** — a description of the ports it observes. It is a *schema*, not data.
`inputs()` returns `self.config['emit']`	The input interface **is** the emit schema. Whatever you declare to emit is exactly what the emitter is wired to read.
`outputs()` returns `{'results': 'node'}`	One output port, `results`, carrying a **handle** — a reference to what was accumulated, not the data itself.
`update(state)` records, returns `{}`	Each tick it persists the observed `state` and returns an **empty delta**, so it can never feed state back into the run.

The empty return is the discipline. A Step that returned a non-empty delta would be a process-in-disguise, mutating the model it is supposed to be passively recording. An emitter returns `{}` every tick — it is a pure sink.

Why `results` is not written every tick

The `results` port is produced at **completion**, by `finalize()`, not by `update()`. Results are meaningful once — at the end of a run — and writing them each tick would re-fire every downstream consumer on every tick. `finalize()` returns `{'results': self.results(...)}`; `update()` stays silent. This is why an emitter can sit in a step network as an ordinary producer without flooding it.

The results handle

`finalize()` hands back an `EmitterResults` — a durable **reference** to what the emitter holds, carrying only the emitter's address, its path, a record count, and any context a consumer needs to interpret it. The bulk data stays where it is until someone calls `.resolve()`:

```
handle = emitter.results()          # cheap: a reference, no data copied
rows = handle.resolve()             # pulls (and memoizes) the accumulated
history
```

The point of the handle is that a downstream analysis or report-card Step can depend on the emitter as a normal producer/consumer edge — and it "cannot tell a live emitter from a pulled cache artifact," because both answer the same `kind` / `context` / `resolve()` protocol. That symmetry is what lets a study **pull** a cached run or **compute** a fresh one through the same wiring. (`EmitterResults.kind == 'trajectory'`.)

The built-in emitters

Three emitters ship **inside** process-bigraph with no extra dependencies. Reach for these first — they cover most of what you need while iterating.

Emitter	Address	Sink	Reach for it when…
`ConsoleEmitter`	`local:ConsoleEmitter`	stdout	you want to *watch* a run print each tick while debugging
`RAMEmitter`	`local:RAMEmitter`	in-memory list	short runs; analysis in the same Python process
`JSONEmitter`	`local:JSONEmitter`	one JSON-Lines file per run	small-to-medium runs you want to keep on disk

They differ only in where `update(state)` puts the row. `ConsoleEmitter` prints it; `RAMEmitter` appends a `tree_copy` of it to `self.history`; `JSONEmitter` appends one JSON object per line to `history_<simulation_id>.json`. All three expose the **same** `query(paths=None)` API, so downstream plotting and analysis code never has to know which one produced the trajectory.

> **RAMEmitter has two knobs worth knowing**
>
> `subsample: N` records only every *N*th tick (default `1` = every tick), and `max_len: N` bounds `history` to the most recent *N* rows as a ring buffer (default `None` = unbounded). Both keep memory in check on long or heavy runs without distorting the time axis — each row still carries its true `global_time`.

Heavier sinks live in `viva-emitters`

The database- and array-backed emitters were **extracted** out of process-bigraph into a focused sibling package so they can carry their own optional heavy dependencies without forcing them on every user:

Emitter	Sink	Extra
`SQLiteEmitter`	one `.db` file, rows keyed by `simulation_id`	`viva-emitters[sqlite]` (stdlib only)
`ParquetEmitter`	hive-partitioned Parquet	`viva-emitters[parquet]` (duckdb, polars, …)
`XArrayEmitter`	a zarr store (`runs.<id>.zarr`)	`viva-emitters[xarray]`

`process_bigraph.emitter` **re-exports** `SQLiteEmitter` , `ParquetEmitter` , and their retrieval helpers lazily — so `from process_bigraph.emitter import SQLiteEmitter` keeps working as long as the sibling package is installed. Install both backends at once via the `process-bigraph[emitters]` extra.

Package name: `viva-emitters` **(formerly** `pbg-emitters` **)**

The sibling package completed the `pbg → viva` rename. The canonical distribution is `viva-emitters` and its import package is `viva_emitters` — which is exactly what the re-export shim in `process_bigraph/emitter.py` imports (`import viva_emitters`). The old names still work: `import pbg_emitters` resolves to `viva_emitters` through a back-compat shim that emits a `DeprecationWarning` , so any older install instructions or docstrings that say `pbg-emitters` point at the same package. Prefer `viva_emitters` ; verify against whatever is installed if in doubt. *(Verified against* `viva-emitters` *0.2.3.)*

Two caveats to keep straight: `XArrayEmitter` **is not in the process-bigraph re-export set** — only `SQLiteEmitter` , `ParquetEmitter` , and their retrieval helpers are (see `_VIVA_REEXPORTS` in `process_bigraph/emitter.py`) — so import it from the emitters package directly, not from `process_bigraph.emitter` . And the `runs.<id>.zarr` filename is a run-store *convention* (how the study spine names a run store): `XArrayEmitter` itself takes an `out_uri` config key and writes wherever you point it, with no `runs.<id>.zarr` literal in its source — confirm the exact path against your config, not this table.

Wiring an emitter

You almost never write an emitter's step spec by hand. Two helpers build it for you.

Inline, with `emitter_from_wires`

The common case: declare the emitter alongside your processes when you build the composite. `emitter_from_wires(wires)` takes a dict of `{name: path}` wires and returns a ready `step` node — it derives the `emit` schema from the wires and sets the inputs to them:

```
from process_bigraph import Composite, Process, allocate_core
from process_bigraph.emitter import emitter_from_wires, gather_emitter_results

class Grow(Process):
    config_schema = {'rate': 'float'}
    def inputs(self):  return {'level': 'float'}
    def outputs(self): return {'level': 'float'}
    def update(self, state, interval):
        return {'level': state['level'] * self.config['rate'] * interval}

core = allocate_core()
core.register_link('Grow', Grow)

composite = Composite({'state': {
    'level': 1.0,                                   # a shared store
    'grow': {'_type': 'process', 'address': 'local:Grow', 'config': {'rate':
0.5},
             'interval': 1.0, 'inputs': {'level': ['level']}, 'outputs':
{'level': ['level']}},
    # record global_time and level every tick into a RAM emitter
    'emitter': emitter_from_wires({'level': ['level'], 'time':
['global_time']}),
}}, core=core)

composite.run(5.0)
print(gather_emitter_results(composite))
# {('emitter',): [{'level': 1.0, 'time': 0.0}, {'level': 1.5, ...}, ...
{'level': 7.59, 'time': 5.0}]}
```

`emitter_from_wires(wires, address='local:RAMEmitter', subsample=1)` defaults to the RAM backend; pass `address='local:ConsoleEmitter'`, `'local:JSONEmitter'`, `'local:SQLiteEmitter'`, and so on to change the sink. The wires you pass **are** the filter: the emitter only ever sees state you wired in, so unwired state is never copied.

After the fact, with `add_emitter_to_composite`

When you have a composite someone else built — or you want to observe *every* wired variable without listing them by hand — attach an emitter to the already-built object and let it rebuild the step network:

```
from process_bigraph.emitter import add_emitter_to_composite,
gather_emitter_results

# 'all' observes every non-process, non-step port in state
composite = add_emitter_to_composite(composite, core, emitter_mode='all')
composite.run(10.0)
```

`emitter_mode` accepts `'all'` (every valid input port in state — the default), `'none'` (only `global_time`), or `{'paths': [['Env', 'x'], 'target']}` (an explicit list). It always adds `global_time`, so every trajectory has a time axis.

The dotted edges are the discipline made visible: state flows **into** the emitter, and the only thing that flows out is a handle. No arrow ever runs from the emitter back into a store.

Reading the trajectory

Two entry points, depending on whether you hold the emitter or the composite.

Per-emitter — `emitter.query()`. Every emitter exposes the same `query(paths=None)`, returning a list of per-tick dicts. Pass paths to project only the wires you want:

```
emitter = composite.state['emitter']['instance']

history      = emitter.query()                          # full history, one dict
per tick
times_only   = emitter.query([['time']])                # just the clock
x_and_target = emitter.query([['x'], ['Env', 'target']])# two named slices
```

All emitters — `gather_emitter_results(composite)`**.** Finds every `Emitter` instance in the composite and returns `{path: history}`:

```
from process_bigraph.emitter import gather_emitter_results

results = gather_emitter_results(composite)
for path, history in results.items():
    print(path, len(history), 'ticks')

# drive each emitter's query independently:
results = gather_emitter_results(composite, queries={('emitter',): [['Env',
'x']]})
```

> **Two places to filter, and which to prefer**
>
> You can filter **at emit time** (the wires you pass to `emitter_from_wires` — the emitter never even sees unwired state) or **at query time** (`emitter.query(paths)` — the full tree was stored, you read a slice back). For long runs prefer emit-time filtering: less memory, smaller files, smaller database rows.

Keeping runs on disk

`RAMEmitter` discards everything when the Python process exits. For runs you want to keep, `SQLiteEmitter` appends one row per tick into a single `.db` partitioned by `simulation_id`, and its retrieval helpers take **only a db path** — so you can analyse a run weeks later without reconstructing the composite or importing the original processes:

```
from process_bigraph.emitter import list_simulations, load_history,
load_simulation_metadata

db_path = './out/history.db'
for sim in list_simulations(db_path):
    print(sim['simulation_id'], sim['name'], sim['step_count'])

history = load_history(db_path, 'run-2026-04-14-001')            # same shape
as query()
only_x  = load_history(db_path, 'run-2026-04-14-001', paths=[['x']])
meta    = load_simulation_metadata(db_path, 'run-2026-04-14-001') # when it ran,
its config
```

Useful config keys on the durable emitters: `file_path` / `db_file` (where), `simulation_id` (defaults to a UUID), `subsample: N` (keep every *N*th tick), and `batch_size: N` (buffer *N* rows per transaction to amortise fsync on high-frequency runs).

Writing a custom emitter

An emitter is "just a `Step` subclass with an `inputs()` method that returns what to observe and an `update(state)` method that records it." Subclass `Emitter`, keep the `emit` key in your `config_schema`, persist in `update`, and return the list-of-dicts shape from `query`:

```
import csv
from typing import Dict
from process_bigraph.emitter import Emitter

class CSVEmitter(Emitter):
    '''Append each tick to a CSV file.'''
    config_schema = {
        **Emitter.config_schema,                        # keep the 'emit' key
        'file_path':  {'_type': 'string', '_default': 'history.csv'},
        'fieldnames': {'_type': 'list[string]', '_default': None},
    }

    def __init__(self, config, core):
        super().__init__(config, core)
        self.file_path = config.get('file_path', 'history.csv')
        self.fieldnames = config.get('fieldnames') or list(self.inputs().keys())
        self._fh = open(self.file_path, 'a', newline='')
        self._writer = csv.DictWriter(self._fh, fieldnames=self.fieldnames)
        if self._fh.tell() == 0:
            self._writer.writeheader()

    def update(self, state) -> Dict:
        self._writer.writerow({k: state.get(k) for k in self.fieldnames})
        self._fh.flush()
        return {}                                       # empty delta: stays
read-only

    def query(self, query=None):
        with open(self.file_path) as f:
            return list(csv.DictReader(f))
```

Register it under an address and use it exactly like any built-in:

```
core.register_link('CSVEmitter', CSVEmitter)
# emitter_from_wires({...}, address='local:CSVEmitter')
```

The contract to satisfy, in one list:

- Accept `emit` in `config_schema` (inherit from `Emitter.config_schema`) — it is the schema of observed state.
- Implement `update(state)` to persist; return `{}` to stay read-only.
- Implement `query(paths=None)` to return a list of per-tick dicts, so it plugs into existing plotting and analysis code.
- If live process/edge objects can be wired into your state, reuse `process_bigraph.emitter.tree_copy` — it strips `Edge` instances before persistence, which is exactly what `RAMEmitter` and `SQLiteEmitter` do.

Where emitters sit in the bigger picture

An emitter is the **durable phase boundary** of a study. Phase one is the temporal simulation; the emitter writes the trajectory; phase two — the analyses, visualizations, and report cards that grade the run — are all Steps that read the emitter's `results` handle. That is why Studies tie each named readout to an exact store path: the study's *emit contract* is a promise about what its emitter records, and the verdict is computed from what actually came out. Getting the data out, cleanly and without letting it leak back into the model, is the seam the whole agentic spine stands on.

Next: Templates & draft processes

composite draft-process template

Templates & draft processes

Core concepts ends on a claim that sounds too tidy to be true: a composite, a template, a study, and an investigation are all *the same kind of thing* — a typed document — and they differ in structure, not in execution machinery. This chapter is where that claim earns its keep. It is about interface-first modeling: designing a document with the mechanism deliberately left out, and filling it in later.

There are two ways to leave a mechanism out, and they are worth keeping apart:

- A **site** is an empty *hole* — a place where a whole composite, process, or value has to plug in before the document can run.
- A **draft process** is a present-but-*inert* node — the interface is there, wired and visible, but it carries no dynamics, so it no-ops when stepped.

A site makes a document refuse to run; a draft lets it run and simply does nothing. Both let you commit to the *shape* of a model before you commit to its *behaviour*.

On this page

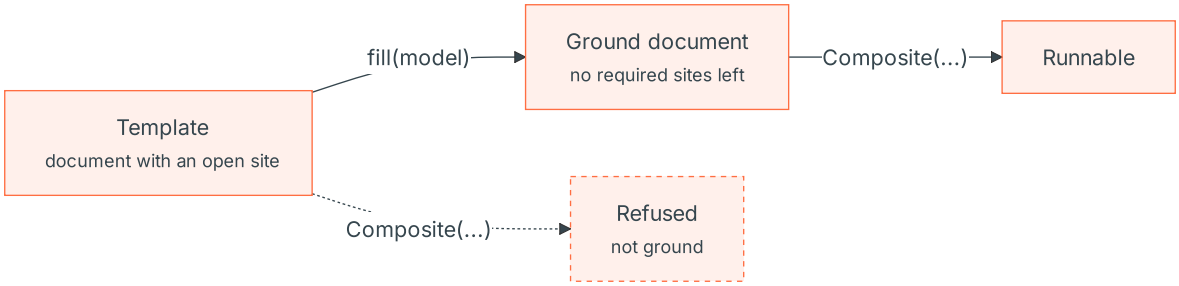
Assumes Composites & wiring, Core concepts. · **You'll learn** sites versus draft processes, the fill operation, the `is_ground` law, and how to model interface-first before committing to behaviour.

Sites, fill, and ground

The framework collapses a lot of apparent machinery into three ideas — one object, one operation, one law:

One object (a typed **document**), one operation (**fill** its sites), one law (`is_ground` — it runs iff no unfilled required site remains).

A **template** is just a document that still has holes. The holes are **sites**, written in the document as `{"_type": "site", ...}`; each site records the *face* (the port interface) a filler must present. The one operation on documents is **fill** — substitute a value or a sub-composite into a site. And the one law is **groundness**: `Composite` refuses to build a document with any open *required* site, because "an open site is a hole where a process should be."



The primitives live in **bigraph-schema** — this is the layer that owns "what a document *is*." Filling and the groundness check are real, shipped functions:

Concept	Where it lives	What it is
a **site** (place-graph hole)	`bigraph_schema.schema` (`Site`), re-exported through `assembly`	a hole in the place graph a filler substitutes into
`fill`	`Core.fill(schema, state, ...)`	substitute fillers into open sites; incremental
`is_ground`	`bigraph_schema.assembly.is_ground(schema)`	true iff no sites remain and all ports are wired

Verified against code

`is_ground` is a real function in `bigraph_schema/assembly.py` ("no Sites, all ports wired"); `Core.fill` is a real method in `bigraph_schema/core.py`; the `Site` place-graph hole is a real type defined in `bigraph_schema/schema.py` and used throughout `assembly.py` (from which it is importable). process-bigraph's `templates.py` imports `fill_sites` from `bigraph_schema.assembly` and builds on all three. This is the machinery the design docs describe as "shipped in bigraph-schema" — the site/fill/ground layer is current API, not a plan.

Legacy vocabulary

Older design documents call sites **slots**, and call filling them **bind** or **reify**. Prefer **site / fill / ground**; treat *slot* / *bind* / *reify* as synonyms when you meet them in older material.

Building and filling a template

process-bigraph wraps the bigraph-schema primitives in a small template toolkit (`process_bigraph/templates.py`) so you can read the state of a document and fill it. The functions are thin and honest:

Function	What it does
`open_sites(document)`	the paths of every site still open
`required_open_sites(document)`	the open sites with no `_default` — the ones that block a build
`is_ground_document(document)`	`not required_open_sites(...)` — the runnable predicate
`fill_template(core, template, bindings)`	fill sites; unbound sites stay open (filling is incremental)
`template_document(core, template, bindings)`	fill **and render** the document `Composite` consumes; raises, naming any required site left unfilled

Here is the whole arc — a "study" template that fixes its analysis and leaves the *model* as a hole, then fills the hole with a conforming composite so it becomes ground and runs:

```
from process_bigraph import Composite, Step, allocate_core
from process_bigraph.templates import open_sites, is_ground_document,
template_document

class ReportCard(Step):                                  # a fixed downstream
verdict
    config_schema = {'threshold': 'float'}
    def inputs(self):  return {'level': 'float'}
    def outputs(self): return {'verdict': 'string'}
    def update(self, state):
        return {'verdict': 'pass' if state['level'] >= self.config['threshold']
else 'fail'}

core = allocate_core()
core.register_link('ReportCard', ReportCard)
core.register_link('Grow', Grow)                         # the Grow process from
the emitters chapter
MODEL_FACE = {'_type': 'link', '_inputs': {'level': 'float'}, '_outputs':
{'level': 'float'}}

# analysis fixed; the model is a HOLE with a declared face
template = core.access({'study': {
    'level': 1.0, 'verdict': 'string',
    'model':  {'_type': 'site', '_sort': MODEL_FACE},
    'report': {'_type': 'step', 'address': 'local:ReportCard', 'config':
{'threshold': 2.0},
               'inputs': {'level': ['level']}, 'outputs': {'verdict':
['verdict']}}}})

print(open_sites(template), is_ground_document(template))   # [('study',
'model')] False

def model(rate):                                         # any conforming composite
fits the hole
    return core.access({'_type': 'process', 'address': 'local:Grow', 'config':
{'rate': rate},
                        'interval': 1.0, 'inputs': {'level': ['level']},
'outputs': {'level': ['level']}})

sim = Composite({'state': template_document(core, template, {'study/model':
model(0.5)})}, core=core)
sim.run(4.0)
print(sim.state['study']['verdict'])                     # pass  (a slower model
would fail)

# template_document(core, template, {}) → ValueError: not ground — required site
'study/model'
```

The last comment is the law in action: hand `template_document` an empty binding and it raises, naming the required site you left open — the same condition `Composite` enforces, reported before construction so you see *which* hole is empty rather than a downstream constructor error.

> The composite is reusable across many questions; the template attaches one question by filling one hole. Separating them means you can re-run the science without rewriting the argument.

This is exactly how the agentic spine works. A **study** is a template with its model site filled; an **investigation** is a document with one site per member study, admitted by filling. process-bigraph even ships `investigation_document(...)`, which fills the members you bind and *prunes* the regions you leave open — so "gating a study" is not a scheduler deciding to skip it, it is simply a site left unfilled. Gating and template binding are the same mechanism. See Studies and Investigations for that layer.

Draft processes

A site is an empty hole. A **draft process** is the complementary idea: a node that is *there but inert*. A `DraftProcess` declares a **contract** — its input and output ports plus a human-readable description of the transformation it is *meant* to perform — but carries **no `update` dynamics**. It inherits the base `Process.update` no-op, so a composite containing a draft still builds and still runs; the draft just contributes nothing.

> A draft lets the model topology — which process connects which stores — be designed and reviewed *before* anyone commits to a mechanism. It never fabricates behaviour it does not have.

That last clause is the whole point. In a framework whose verdicts are computed from what a run actually produced, a placeholder that quietly invented some plausible dynamics would be a lie the evidence layer could not catch. A draft is honest by construction: stepped, it returns `{}`, and it announces itself as unfinished. This is the meta-modeler's move made concrete: declare what a part

exposes — its ports and intent — before committing to how it works, since an interface is a concrete, testable target even while the mechanism behind it stays open.

`@draft_process`

You define a concrete, registrable draft with the `@draft_process` decorator. It injects the ports and contract onto a bare `DraftProcess` subclass:

```
from process_bigraph import DraftProcess, draft_process

@draft_process(
    name="PTH secretion",
    inputs={'ca_sense': 'float'},
    outputs={'pth_out': 'float'},
    contract={'summary': 'senses serum Ca, secretes PTH',
              'senses': 'calcium', 'makes': 'PTH'})
class PTHSecretion(DraftProcess):
    pass

core.register_link('PTHSecretion', PTHSecretion)
print(PTHSecretion({}, core=core).describe())
# DRAFT — senses serum Ca, secretes PTH  ·  makes: PTH  ·  senses: calcium  ·
status: draft - no update dynamics yet
```

There is no `update` on the class — deliberately. `describe()` renders the whole contract prefixed with `DRAFT —`, and the decorator stamps `status: "draft - no update dynamics yet"` into the contract automatically. Because a module-scope `DraftProcess` subclass is discovered and auto-registered by a workspace's `build_core()` (the same walk that registers every Process and Step), a draft **shows up in the Workbench dashboard by name, with its ports and contract, marked DRAFT** — with no extra wiring. Replace it with a real `Process` once the mechanism is committed, and nothing else in the composite has to change.

Verified against code — `DraftProcess` is a shipped primitive

`DraftProcess` and `@draft_process` are real, exported from `process_bigraph` (`process_bigraph/draft_process.py`; re-exported in `__init__.py`). The decorator signature is `@draft_process(*, name, inputs, outputs, contract)` — all keyword-only. Registry tooling reads a draft *uninitialized*: it calls `inputs()` / `outputs()` on the class and `describe()` on a bare instance, which is why every method reads class attributes and needs no configured instance.

Three "blanks" are easy to confuse — keep them apart:

Blank	State of the document	Fills it with
site	not runnable until filled (a hole)	`fill` — a value or sub-composite
draft process	runnable; the node no-ops	replacing it with a real `Process`
`config` / params	runnable; the node runs	supplying parameter values

A site is an *empty hole*; a draft is a *present-but-inert node*; a config value fills a node's *parameters*, not a hole and not a node.

The compiler view

Draft processes point at something larger than a scaffolding trick. Read the ecosystem's design notes and a picture emerges of the framework as a **compiler** — from what a biologist *means* to what an engine can *run*.

> People describe biological systems by their meaning. Simulators need an executable graph. Between them sits a compiler. — *compiling-biology-to-bigraphs*

In that framing (from the conceptual corpus, `compiling-biology-to-bigraphs.pdf`, dossier 05 §4), a `DraftProcess` — a contract with roles, ports, and intent but no

dynamics — is the **source-level semantic model**: the top layer of a small, nanopass-style pipeline. Below it sit a typed reaction-network intermediate representation, then an executable composite (processes, ports, place graph, an emitter), then `Composite.run()`. The step that turns an abstract mechanism into one concrete simulation is read as an **algebraic-effect handler**: one declared effect, many handlers, so a single semantic model can be closed off into a deterministic mass-action realization or a stochastic one — "same meaning, two executable interpretations, agreeing where theory says they must."

The slogan that captures the discipline:

> A process-bigraph process should be *one possible realization* of a biological mechanism, not the definition of that mechanism. — *vivarium_semantic_layer*

> ⚠ **Accuracy note — direction, not a shipped one-click feature**
>
> The site/fill/ground machinery and `DraftProcess` are **shipped code** (verified above). The full "biology → executable" compiler — an elaborator library that turns a semantic `kind` into elementary reactions, a handler registry that fixes how each reaction is simulated, ontology grounding — is a **documented direction**, explored in a working prototype (`littleb-pbg`) and specified in the semantic-layer RFCs, not a one-click feature in the current package. The load-bearing pieces you can rely on today are the two on this page: leave a hole (**site**) or leave a mechanism undeclared (**draft**), and fill it in when the science is ready. Treat the compiler itself as where the interface-first workflow is *heading*.

The same discipline scales past a single process. Followed all the way up, a whole cell reads as a self-maintaining organization whose metabolism, containment, and replication processes each present an interface the others fill — supplying the components, gradients, and boundary that keep the network running — and building such a model means repeating the one move: declare an interface, realize it with a conforming submodel, until the organization closes on itself.

The through-line is the one this chapter opened with, now with teeth: a template is a document with holes, a draft is a document with an inert node, and the same act — filling the hole, supplying the mechanism — turns designed structure into executable behaviour without ever letting the framework pretend a mechanism exists before it does.

Next: Workspaces & the Workbench

PART 3

Investigate

The agentic spine — how runs become auditable scientific evidence. This part is for scientists: it frames each run as a Study with a pass/fail bar, rolls studies up into gated Investigations, and computes verdicts you can trust rather than assert.

Workspaces & the Workbench

The directory that *is* the model, and the AI-free server that turns it into a git-backed research notebook.

Studies

One question, one emit-contract, and one pass/fail bar wrapped around a composite.

Analyses, visualizations & report cards

The machinery that turns a wall of numbers into figures, derived tables, and graded scorecards.

Investigations

A gated DAG of studies that accumulates into a defensible claim under one research question.

Rigor & the evidence engine

The deterministic pipeline that computes a verdict from a run and writes the whole chain back for audit.

Working with AI agents

How the `/viva-*` skills drive the Workbench over plain HTTP, and how the work splits between agent and human curator.

Suggested path

Start with **Workspaces & the Workbench** for where the work lives, then **Studies** for the unit of evidence and **Investigations** for how studies combine. **Analyses** and **Rigor** explain how verdicts are produced; **Working with AI agents** shows who drives the loop.

getting-started workbench workspace

Workspaces & the Workbench

Everything in the Investigate half of the guide happens in one place: a **workspace** — a directory that *is* the model — driven by the **Workbench**, an AI-free server that turns that directory into a git-backed research notebook.

> *A workspace is where research happens; the Workbench is the loop turning inside it. The data lives in the workspace, never in the server.*

On this page

Assumes Core concepts. · **You'll learn** the workspace/Workbench split, which way the dependency arrow points, and how every mutating action becomes a git commit.

The one crucial split

The Workbench is *tooling*; the workspace is *data*. They are separate on purpose.

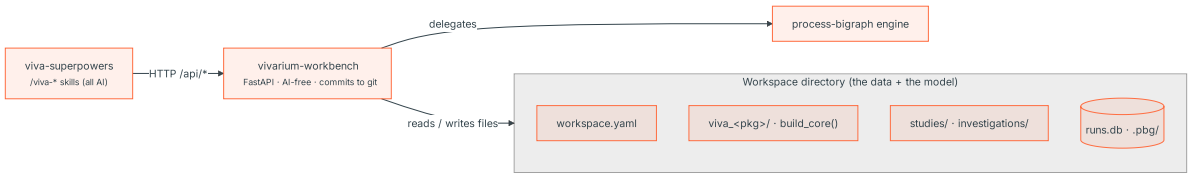
`vivarium-workbench` — the server

A single-process **FastAPI** web app. It has **no database of its own** and **no AI**. It reads and writes plain files in a workspace directory and delegates every simulation to `process-bigraph`. Every mutating action commits to the active git branch, so there is a full audit trail.

the workspace — the data

A git repository holding the model's Python package, its composites, its studies and investigations, its references, and its results. It is the **unit of reproducibility**: clone it, run it, get the same answer. The server runs *inside the workspace's own venv*.

The dependency arrow points one way: the **workspace depends on** `vivarium-workbench` (it is a pip dependency in the workspace's `pyproject.toml`), never the reverse. That is why the server can import the workspace's own package and any installed simulator stacks — it is running in their environment.



> The Workbench is AI-free. All AI capability is packaged as the `viva-superpowers` Claude Code plugin — a set of `/viva-*` skills that drive the Workbench's HTTP API. This keeps the tool auditable and the AI swappable.

Modules vs workspaces

Two nouns are easy to confuse, so pin them down early:

	Module	Workspace
What it defines	processes, composites, runtime-environment dependencies	studies, composites, investigations — with modules installed
Ownership	distributable, versioned, immutable at a version	owned, mutable, *not* itself a unit of distribution
Back-reference	a module has no back-reference to any workspace	a workspace holds a list of modules

A **module** is a `viva-<tool>` / `pbg-<tool>` package that wraps a simulator; a **workspace** installs several of them and attaches science to the composites they provide. A **composite-only repo** (e.g. a bare simulator wrapper) has *no* `workspace.yaml` — just a `pyproject.toml`, a `viva_<slug>/` package, and `tests/` — and is pulled into a workspace through `workspace.yaml`'s `imports`. You can promote such a repo into a workspace in place (see scaffolding, below).

What a workspace looks like on disk

A workspace is a directory with a `workspace.yaml` manifest and a `viva_<pkg>/` Python package that exposes a `build_core()` entry point. Everything else hangs off those two.

```
my-workspace/
├── workspace.yaml                          # the manifest (schema below)
├── viva_<pkg>/                             # the workspace's Python package
│   ├── core.py                             #   build_core() — registers this
repo's processes
│   ├── composites/<id>.composite.yaml      #   the runnable substrate
(JSON/YAML or a generator .py)
│   ├── processes/                          #   Process / Step classes
│   └── visualizations/                     #   Visualization Steps
├── investigations/<slug>/investigation.yaml   # a collection (= git branch =
worktree)
│   └── studies/<slug>/study.yaml           #   studies commonly nest under
their investigation
├── studies/<slug>/                         # (flat layout still resolves for
legacy studies)
│   ├── study.yaml                          #   a question + its narrative spine
│   ├── runs.db                             #   canonical run + outcome record
(SQLite)
│   ├── parquet-runs/<run>/                 #   emitted trajectories (Parquet
hive)
│   └── viz/                                #   rendered figures + report cards
├── references/papers.bib                   # shared bibliography
├── notes/                                  # field records — cleanup PRs must
SPARE these
└── .pbg/                                   # gitignored runtime state (see
below)
    ├── server/server-info                  #   the live server URL every client
reads
    ├── runs/<run_id>/                      #   detached-run request +
observables
    ├── composite-runs.db · artifacts/      #   run metadata + content-addressed
caches
    └── schemas/ · events.jsonl · state.json
```

`.pbg/` — a legacy prefix that stayed

The runtime control directory is still named `.pbg/`, and the global registry still lives at `~/.pbg/`, even in viva-branded workspaces. The `pbg → viva` rename covered skills and packages but deliberately left these paths alone so existing tooling keeps working. See A note on names.

The `build_core()` convention

The workspace package's job is to hand the engine a `Core` — the type-and-process registry (see Schemas, types & state). By convention the package exposes a single `build_core()` function that composes any imported repos' cores and then registers this repo's own processes:

```
# viva_<pkg>/core.py
def build_core(core=None):
    core = compose_import_cores(core)              # pull in imported module
stacks
    register_package_processes(core, f"{__package__}.processes")
    return core
```

> **Accuracy note — `viva_` vs `pbg_`, `build_core()` vs `core.py`**
>
> The ecosystem is mid-migration. **Newer** scaffolds emit a `viva_<pkg>/` package exposing `build_core()`. **Older** docs and workspaces show a `pbg_<pkg>/` package and refer to the registration entry point simply as `core.py`. The *convention* — "the package exposes a `build_core()` that returns a populated `Core`" — is stable; only the package prefix and some prose differ. Prefer the `viva_` spelling.

The `workspace.yaml` manifest

`workspace.yaml` is validated against a Draft-07 JSON schema shipped in the plugin (`viva_superpowers/schemas/workspace.schema.json`).

Required: `schema_version`, `name`, `created` (a date), `plugin_version` (semver).

Commonly used optional keys:

Key	What it declares
`package_path`	the `viva_<pkg>/` directory holding the package
`default_baseline`	pre-filled composite + params + run-knobs for new study baselines
`imports`	a map of imported `pbg-*` / `viva-*` repos a composite instantiates directly (`{source, ref, mode, installed}`)
`observables`	named `{name, store_path, units}` observables the workspace tracks
`visualizations` · `simulations` · `datasets`	declared figures, run recipes, and input datasets
`references_bib` · `references_pdfs`	the shared bibliography and cited PDFs
`server.enabled`	whether this workspace runs a dashboard server
`ui.composite_view`	which renderer draws composite wiring (`loom-explore` — the default — or legacy `bigraph-viz`)
`layout`	optional per-directory relocations (e.g. group `studies/` under `workspace/studies/`)

⚠ **Accuracy note — schema version and the `runtime:` block**

The shipped `workspace.schema.json` pins `schema_version` to the enum `[2, 3]` and does not constrain a top-level `runtime:` block. The concept docs describe a `runtime:` block (`default_emitter: parquet|sqlite` , `subprocess_timeout_s` , `shared_artifacts: [...]`) that migrated workspaces carry; treat it as a documented, schema-tolerated convention rather than a required, schema-enumerated field.

Scaffolding a workspace

The `/viva-workspace` skill (backed by the `viva-scaffold` console script and the `vwb` CLI) creates a workspace in one of **three modes**, chosen by your starting state:

Starting state	Mode	Command
No directory yet, no upstream model repo	**standalone**	`/viva-workspace <name>`
No directory yet, want to branch off an existing repo	**upstream-branch**	`/viva-workspace <name> --upstream owner/repo`
Already inside a git checkout you want to promote	**in-place**	`/viva-workspace <name> --target . --in-place`

- **standalone** clones the **viva-template** scaffold (`vwb scaffold-workspace`), runs `git init`, creates a `.venv` and `uv pip install -e .[dev]`, commits the bootstrap, and registers the workspace in the global catalog (`~/.pbg/workspaces.json`).
- **upstream-branch** clones an upstream model repo, cuts a workspace branch off `origin/main`, applies the scaffolding on top, and commits — the recommended path when you are adding an investigation to code that already exists.
- **in-place** promotes a checkout you already have (skipping any files that already exist), then registers it. This is the right answer for composite-only repos.

> **Template repo naming**
>
> The scaffold source is the **viva-template** repo. The canonical scaffolder (`viva_superpowers/scaffold.py`) defaults its clone URL to `vivarium-collective/viva-template`; the older `/viva-workspace` skill prose still names `pbg-template`, and either `$VIVA_TEMPLATE` or `$PBG_TEMPLATE` (or `--template-source`) overrides it. Same scaffold, two names. After scaffolding, `python3 scripts/lint-workspace.py` should print `workspace lint: OK`.

If you prefer to do it by hand, cloning the template and initializing it is equivalent:

```
git clone https://github.com/vivarium-collective/viva-template my-workspace
cd my-workspace
bash use-this-template-init.sh          # renders the .j2 scaffolding
uv venv && source .venv/bin/activate
uv pip install -e ".[dev]"              # pulls in vivarium-workbench as a
dependency
python3 scripts/lint-workspace.py       # -> "workspace lint: OK"
```

Starting the server

From the workspace root (the directory containing `workspace.yaml`):

```
vivarium-workbench serve --workspace .              # picks a free port, then
serves
vwb serve --workspace . --port 8000 --host 0.0.0.0  # vwb is the short alias
bash scripts/serve.sh                               # convenience shim in a
scaffolded workspace
```

On start the server writes a JSON `server-info` doc to `.pbg/server/server-info` — the file every client and agent reads to discover the server. It carries `port`, `host`, and a ready-to-use `url` (clients parse it with `json.loads`, not as a bare string). **Never hardcode a port.**

```
BASE=$(python3 -c "import json; print(json.load(open('.pbg/server/server-info'))
['url'])")
curl -s "$BASE/api/workspace-manifest"   # one-call situational snapshot — start
here
```

Because the server runs in the workspace's venv, it can import the workspace's `build_core()` and any installed `pbg-*` / `viva_*` simulator stacks directly.

> **Running modes**
>
> Point the server at a remote **viva-api** backend (`VIVA_API_BASE=...`) to submit runs that execute on GovCloud (Ray → AWS Batch → zarr/parquet on S3) and land back as study runs. The same codebase, with writes gated to a small whitelist, is what serves the **read-only online dashboard** — "the workbench with writes gated."

Session-per-tab: one workspace per browser tab

The server can host many workspaces at once. It keeps them from colliding with a simple rule: **a session is pinned to one workspace, per browser tab, for its life.**

- Picking a workspace from the left-rail **workspace switcher** opens it in a **new tab** rather than re-pointing the current one — "one workspace per browser tab."
- Session middleware resolves each request's session to a workspace and sets a **request-scoped** root (a `ContextVar`), so two tabs on two workspaces never interfere.
- **Runs are workspace-scoped, not session-scoped.** A run outlives the session that launched it; another tab on the same workspace sees it in the Runs index.

The side-rail tabs

The left rail is the map of the Workbench. The shipped labels (authoritative) and what each surfaces:

Rail tab	Shows
Resources	the `workspace.yaml` summary — dependencies, references, datasets, scaffolding status
Catalog	the marketplace / module catalog: install and list available processes and composites
Processes	the **Registry** — every Process / Step / Composite the workspace can import
Studies	the Investigations DAG canvas, grouping studies into research arcs
Runs	the Simulations DB / runs index — every run across every emitter backend
Analysis	saved interactive visualizations, 3D viewers, and the Analysis Tools / PTools card

> **Two vocabularies**
>
> An earlier README describes "seven tabs" (Workspace · Registry · Composites · Studies · Investigations · Visualizations · GitHub Branches). That is the *conceptual* model; the shipped rail above is the reorganized, renamed reality. Sections such as the single-study view, the **Composite Explorer**, GitHub Branches, and the audit panel exist but are not top-level rail links — they open from within the tabs above.

Every mutating action in any tab commits to the active git branch in the workspace, so there is a full git audit trail of the investigation as it is built.

The Composite Explorer (loom)

Opened from the Catalog or Composites views (or with `/viva-explore <spec-id>`), the **Composite Explorer** browses, configures, and runs a single composite. Its centerpiece is an embedded **bigraph-loom** state-tree viewer — an interactive rendering of the composite's place graph and wiring, served at `/loom-explore`.

The composite **registry** the Explorer draws from is the union of three sources:

1. the workspace's own package `composites/`,
2. every installed `pbg-*` / `viva_*` distribution's `composites/`, and
3. `@composite_generator`-decorated factory functions discovered across those packages.

A composite is referenced by a dotted id like `v2ecoli.composites.baseline.baseline` — the same id a study points at. From the Explorer you can run a scratch simulation (a detached, durable run), inspect the emitted observables, and promote a configured composite into the catalog. See Composites & wiring for how composites are built.

Golden rules for driving the Workbench

- Read `.pbg/server/server-info` for the URL — never hardcode a port.
- Orient with `GET /api/workspace-manifest` (state) and `GET /api/linkage-index` (graph); trust `GET /openapi.json` for exact shapes.
- **Runs are async** — poll status and check the *result* field, not just HTTP 200.
- Every write is a git commit in the workspace.

Next: Studies

report-card study

Studies

The one sentence

A **Study** wraps one question, one emit-contract, and one pass/fail bar around a Composite.

A composite is reusable across many questions. A study attaches *one* of those questions to it: which composite to run, what to measure, and what "passing" means. An Investigation then lets a dozen such questions accumulate into a defensible claim.

> *Separating them means you can re-run the science without rewriting the argument — and audit the argument without re-reading the code.*

A study is **never compiled into a composite of its own.** It stays *metadata that points at* a composite by a dotted id and reads back what came out. That is the whole design: the composite runs, the study records.

On this page

Assumes Workspaces & the Workbench, Composites & wiring. · **You'll learn** the study/composite split, the three on-disk layers, and the five-phase authoring lifecycle.

Three on-disk layers

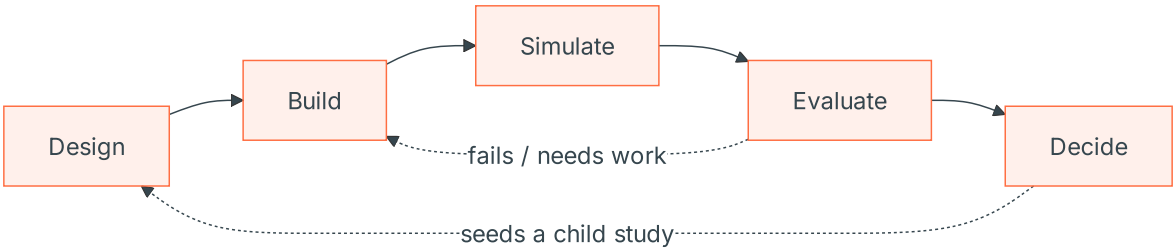
Layer	File	What it is
Investigation	`investigations/<slug>/investigation.yaml`	a named collection of studies = a git branch = a worktree
Study	`studies/<slug>/study.yaml`	one question + its narrative spine (nests under its investigation, or flat for legacy)
Composite	`viva_<pkg>/composites/<id>.composite.yaml`	the runnable process-bigraph document

Read top-down it is an argument; read bottom-up it is a simulation.

The five-phase lifecycle

Authoring a study runs through five phases. They are coarsely sequential and iterative in practice — **Evaluate routinely sends you back to Build.** Each phase *writes a different part of the study spine*:

Phase	Produces	Writes to
Design	the spec: question, gate, run set, tests	`purpose`, `pipeline_gate`, `simulation_set`, `behavior_tests`, `readouts`
Build	executable code: Process classes + composites	`model_change`, `implementation_requirements`, the composite files
Simulate	the runs: trajectories	`runs.db`
Evaluate	the verdict: test results + figures	`outcomes`, `visualizations`, `findings`
Decide	conclusion + follow-ups	`conclusion_verdicts`, discovery implications



The `phase:` field on a study is a capitalized enum — `Design | Build | Simulate | Evaluate | Decide`. An investigation card surfaces its *slowest-phase* member, so one study stuck in Design holds the whole investigation in Design.

Phases vs the study-detail 'acts'

On the study-detail page the same work is grouped into five reading *acts* — **Study** (overview) · **Design** (what you author) · **Evidence** (what came back) · **Assurance** (what grades it) · **Decision**. The load-bearing boundary there: a **report card grades** (→ Assurance › Tests); an **analysis derives** with no verdict (→ Evidence › Analyses). See Analyses, visualizations & report cards.

How a study uses its composite — the five join fields

A study does not rebuild its composite. It *specializes* one through exactly five fields:

1 · Baselines

`baseline[]` (or v4 `conditions.baseline`) — the runnable composite(s). A **non-empty list** so a study can compare several composites side by side: `{name, composite, params}`.

2 · Variants

`variants[]` — **parameter overrides only**: `{name, base_composite, parameter_overrides}`. No structural or initial-state edits; `base_composite` must name an existing baseline entry.

3 · Simulation set

`simulation_set[]` — the run recipe (base model, perturbation, condition, seeds, duration). In v4 it *replaces* the separate v3 `variants:` + `interventions:` lists.

4 · Readouts — the emit contract

`readouts[].store_path` — ties a named observable to the **exact emission path** in the composite's output. This is the contract that says *what the run must emit* to be gradable.

5 · Behavior tests — the verdict

`behavior_tests[]` — a machine-checkable spec: a `measure` (how to extract a number from run history) plus a `pass_if` (op + value/range). This is what turns a trajectory into a pass or fail. A test may carry a `variant:` sub-field to scope it to one named variant.

Running a baseline (`POST /api/study-run-baseline`) resolves the baseline id, **builds that composite in-process**, merges the params, runs it, and records the trajectory in `runs.db`. The study itself is never compiled.

Derive-on-read: verdicts are computed, never asserted

This is the single most important rule of the study spine, and it is worth internalizing:

> A study's conclusion is **computed from its evidence, not asserted.** You never write `status: pass` by hand. A per-test pass/fail pill is derived from the latest run's `outcomes[test].result` in `runs.db`. This is what stops a study from *claiming* a result it never produced.

The spine keeps **authored** intent and **computed** evidence in *parallel* fields, and a `diverges_from_authored` flag is the dashboard headline when they disagree:

Authored (a human writes)	Computed (code fills, on read)
`runs[].outcomes`	`runs[].computed_outcomes`
`gate_status`	`pipeline_gate.gate_evaluator`
`executive.verdict`	`.computed_acceptance`
`finding.statement` / `.summary`	`finding.evidence` / `.expected` / `.provenance`

So a test authored `status: passed` but backed by **no** run outcome renders as a pending pill (**pending**), not a pass. A study renders **Ran · Tests N✓ · Passed** only when `runs[].outcomes` actually back the results.

The rolled-up **verdict** (`study_verdict.roll_up_verdict()`) aggregates outcomes and prerequisites into a code-computed gate state — passed iff `fail == 0 and pass > 0`:

passed **failed** **needs_calibration** **blocked** **not_started**

It writes a *parallel coded slot* and never overwrites the authored gate.

> Because the study is typed data, hardening it — filling gaps, citing bands, tightening a claim — is a transformation an agent can apply and a human can verify, not a vibe.

"Five phases" vs "six status axes"

Two similar-looking lists travel together in the docs. Keep them distinct:

Five **phases** — a sequence

`Design → Build → Simulate → Evaluate → Decide`. *Where you are* in authoring the study. One value at a time; the `phase:` field.

Six **status axes** — independent dimensions

`design · implementation · simulation · evaluation · gate · expert_review`. Each is a *separate*, nullable axis, so "we wrote it" is never confused with "it ran" or "it holds up."

The six axes replace the single coarse `status:` field for new specs, each with its own enum (e.g. `simulation_status: not_run · running · ran · failed`; `gate_status: blocked · needs_calibration · passed · failed`). The headline pill on the study page

prefers `gate_status` when set. They are additive — a legacy study with only `status:` renders exactly as before.

The `study.yaml` narrative spine (v4)

A v3 `study.yaml` is organized into **eight canonical sections** plus two cross-cutting fields. The v4 schema adds a *second layer* of optional **narrative-spine** fields — the shape the DnaA-replication investigation evolved through use, now promoted to canonical-optional. A v3 spec validates unchanged against v4; new scaffolds land the v4 fields commented in as TODO placeholders.

The eight canonical sections:

#	Section	YAML field(s)
1	Purpose	`purpose:` (`question` / `mechanism` / `expected_outcome`)
2	Pipeline Gate	`pipeline_gate:` (`prerequisites` / `enables` / `proceed_condition`)
3	Simulations	`simulation_set:` (replaces v3 `variants:` + `interventions:`)
4	Build	`model_change:` + `implementation_requirements:`
5	Readouts	`readouts:` (replaces v3 `observables:`; each carries `status`)
6	Tests	`behavior_tests:` (replaces v3 `expected_behavior:`)
7	Limitations	`limitations:`
8	References	`bibliography:`

Cross-cutting: `key_assumptions:` (pairs with Build) and `conclusion_logic:` (`if_primary_tests_pass:` / `if_primary_tests_fail:`, pairs with Tests).

The v4 narrative spine, four layers (all optional; the six ★ fields are authored first and render at the top of the report):

- **Executive** — `title`, `claim`, `confidence` (`Accepted | Investigating | Planned | Refuted`), `runtime`, ★ **`report:`** (verdict / confidence / evidence_quality / key_metrics), ★ **`study_card:`** (one-paragraph card).
- **Framing** — ★ **`question:`** + `assumptions:`, ★ **`conditions:`** (the grouped v4 alternative to top-level `baseline:` + `variants:`, whose `model_settings[]` is the source of truth for tunable parameters — current value + default + literature range + cite), `enforced_params:` (values each run is *required* to apply).
- **Validation** — ★ **`behavior_tests:`**, ★ **`readouts:`** (`store_path` = the emit contract), `biological_summary:` (textbook prose), `literature_anchors:`.
- **Implementation + decisions** — `model_change`, `implementation_requirements`, `design_pivot_required:` (open decision points), ★ **`conclusion_verdicts:`** — a three-track block `{regression_compatibility, biological_validation, explanatory_gain}`, each `{result, basis}`, so a study can be "PASS on regression but MIXED on biology" instead of one forced boolean.

> ⚠ **Accuracy note — the schema version is in flux**
>
> Study specs exist as **v2** → **v3** → **v4**, migrated on load. This chapter presents **v4**; the `schema_version:` field selects the shape (and, at v4, the presence of a top-level `conditions:` block disambiguates two v4 variants). Set `schema_version: 4` to opt into the v4 reserved field names. A handful of names are **reserved** at v4 and were renamed from v3 to avoid collisions — `references:` → `bibliography:`, `tests:` is reserved (so the field is `behavior_tests:`). When in doubt, read a live `study.yaml` in a real workspace rather than inventing a field.

A realistic `study.yaml` skeleton

Faithful to the shipped shape; values truncated to placeholders.

```yaml
schema_version: 4
name: dnaa-01-expression-dynamics       # the slug is the technical id
title: DnaA expression dynamics          # human display name (v4 spine)
created: '2026-08-14'
phase: Evaluate                          # Design | Build | Simulate | Evaluate
| Decide

# --- six independent status axes (replace the coarse `status:`) ---
design_status: approved
implementation_status: complete
simulation_status: ran
evaluation_status: evaluated
gate_status: needs_calibration
expert_review_status: requested

# 1 · PURPOSE
purpose:
  question: |
    Does DnaA-ATP fraction cycle within its literature band across generations?
  mechanism: |
    Titration of DnaA by datA and the chromosomal DnaA boxes.
  expected_outcome: |
    DnaA-ATP fraction stays within 0.2-0.5, oscillating once per division.

# 2 · PIPELINE GATE
pipeline_gate:
  prerequisites: []                      # parent study slugs (bare slug or
{study, condition})
  enables: [dnaa-02-initiation]
  proceed_condition: tests-passed

# 3 · SIMULATIONS  (v4 grouped form)
conditions:
  baseline:
    composite: v2ecoli.composites.baseline.baseline
    params: {}
  variants:
    - name: high-datA
      base_composite: baseline
      parameter_overrides: { datA_copies: 2 }
  model_settings:
    - name: dnaa_synthesis_rate
      current: 0.85
      default: 1.0
      range: [0.5, 1.5]
      units: 1/s
      cites: [Hansen2018]

# 4 · BUILD
```

```yaml
model_change:
  - kind: change_config
    path: processes.dnaa
implementation_requirements: []

# 5 · READOUTS  (the emit contract)
readouts:
  - name: dnaa_atp_fraction
    store_path: listeners.dnaa.atp_fraction
    status: available
    units: dimensionless

# 6 · TESTS  (measure + pass_if -> the verdict)
behavior_tests:
  - name: atp-fraction-in-band
    classification: primary
    measure:
      kind: generation_average           # the discriminating sub-key is `kind`
      path: listeners.dnaa.atp_fraction
    pass_if: { op: in_range_every_generation, low: 0.2, high: 0.5 }
    cites: [Boesen2024]

conclusion_logic:
  if_primary_tests_pass: |
    DnaA titration reproduces the observed cycling band.
  if_primary_tests_fail: |
    Revisit the datA titration parameters before advancing.

# --- v4 executive spine (authored last, renders first) ---
report:
  verdict: passing-with-caveats          # free-form; computed acceptance sits
alongside
  confidence: medium
  evidence_quality: literature-matched
  key_metrics: [dnaa_atp_fraction]
conclusion_verdicts:
  regression_compatibility: { result: PASS, basis: "matches v1 baseline" }
  biological_validation:    { result: MIXED, basis: "band met in 6/7
generations" }
  explanatory_gain:         { result: POSITIVE, basis: "resolves initiation
timing" }

# 7 · LIMITATIONS
limitations: |
  Single-seed; ensemble replication pending.

# 8 · REFERENCES
bibliography:
  - Boesen2024
```

```
  - Hansen2018

# provenance — outcomes are written by the run, never by hand
runs: []                                  # {run_id, outcomes{...}} — filled from
runs.db
findings: []
```

> **You author fields; you do not author verdicts**
>
> The `runs[].outcomes`, `computed_outcomes`, the rolled-up gate verdict, and the per-finding `evidence` / `provenance` slots are all **derive-on-read**: code fills them from the latest run in `runs.db`. Your job is the *authored* side — the question, the tests, the bands, the prose. See Rigor & evidence for how the computed side is graded.

Where studies fit

- Build the composite a study runs → Composites & wiring.
- Grade the runs, render figures and report cards → Analyses, visualizations & report cards.
- Group studies into a gated argument → Investigations.

Next: Analyses, visualizations & report cards

report-card study visualization

Analyses, visualizations & report cards

A simulation produces a wall of numbers. This chapter is about the machinery that turns that wall into something a person — or an agent — can read and trust: the **figures**, the **derived tables**, and the **graded scorecards** a study emits after it runs. All three are built from the same primitive you already met in Core concepts: a **Step** — a non-temporal edge that fires when its inputs are ready.

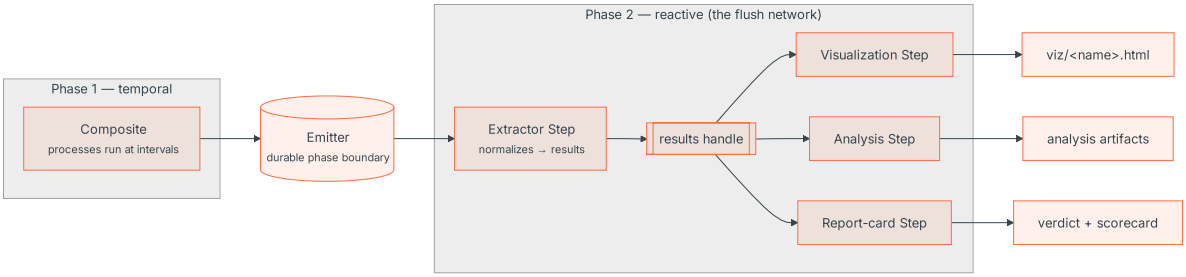
> *Emitters, analyses, visualizations, and report cards are all Steps. The clock never drives them; the arrival of a completed run does.*

On this page

Assumes Studies, Composites & wiring. · **You'll learn** the two-phase study, why emitters, analyses, visualizations, and report cards are all Steps, and how the flush network fires.

The two-phase study

A study is not one simulation and then, separately, some plotting code you run by hand. It is a single **two-phase process bigraph**, and the seam between the phases is the most important idea in this chapter.



- **Phase 1 is temporal.** The composite's Processes advance the state tree at their declared intervals — an ODE integrator, an FBA solve, a stochastic step. This is the science running.
- **The emitter is the durable phase boundary.** As the run proceeds, an Emitter Step records the wired state each tick into a durable sink — SQLite `runs.db`, a Parquet hive, or a Zarr store. Everything before the emitter is ephemeral in-memory state; everything after it reads from disk. The run can finish, the server can restart, and the evidence is still there.
- **Phase 2 is reactive — the flush network.** Once the emitter has written, an **extractor Step** normalizes that emitter output into a single `results` handle, and a DAG of downstream Steps reads that handle and writes artifacts: visualizations write HTML, analyses write figures and tables, report cards write graded verdicts.

Because Phase 2 is a Step DAG, it obeys the one law of Steps: it settles to convergence whenever the state it depends on changes. Re-run the study and the whole flush network re-fires against the new run. Nothing in Phase 2 is a script you remember to launch.

> **Design vs shipped**
>
> The clean "extractor Step → single typed `results` handle → uniform artifact DAG" picture is the framework-unification target (the two-phase **Study composite**, Layer 1). Much of it is real today: runs land in a durable emitter store, the workbench renders visualizations and runs analyses over that store after each run (`lib/composite_flush.py`, the post-run hooks), and data-driven report cards read run records through a `ResultsStep` handle that exposes a DuckDB view literally named `results`. Where the shipped path still uses per-artifact rendering hooks rather than one first-class extractor node, this guide flags it. The *mental model* — durable boundary, then a reactive network of artifact Steps — is stable; verify exact wiring against the current code.

Authoring a visualization

A visualization is a **Step that emits HTML**. You do not have to write a class by hand — the `/viva-viz` skill generates one from a natural-language description into your workspace package. What it writes is a single decorated function.

```python
from process_bigraph.visualization import as_visualization

@as_visualization(
    inputs={'dnaa_atp_fraction': 'list[float]', 'time': 'list[float]'},
    name='DnaATrajectory',
    demo={'dnaa_atp_fraction': [0.31, 0.42, 0.28], 'time': [0.0, 1.0, 2.0]},
)
def update_dna_a_trajectory(state):
    # build an interactive Plotly figure from the wired inputs …
    return {'html': fig.to_html(full_html=False)}
```

The pieces that matter:

Element	Rule
Function name	must start `update_` — the decorator turns it into a Step class
`inputs=`	bigraph-schema **type strings** (`'float'`, `'list[float]'`, `'list[list[float]]'`, `'string'`) — this is the wire contract
`name=`	the CamelCase class the dashboard surfaces; its canonical address is `local:<ClassName>`
`demo=`	realistic synthetic state so the dashboard can render a preview before any run exists
Return	a dict with an `'html'` key

`@as_visualization` is one of three function-to-class decorators (`as_step`, `as_process`, `as_visualization`) provided by `process_bigraph`; the decorator stamps `__pb_kind__` / `__pb_aliases__` metadata so the class surfaces cleanly. You never touch `__init__.py`

— discovery walks the package and auto-registers every `Step` subclass (see Composites & wiring).

> **Decorator vs subclass — a real internal tension**
>
> `/viva-viz` emits the `@as_visualization` **decorated-function** form. The `visualizations` convention doc, however, prefers subclassing `Visualization` directly (a real `Step` with an `html` output port, wireable into composites) and marks the decorator as legacy. Both work and both are in the codebase; v2ecoli uses subclasses. Treat them as two spellings of one idea — an HTML-emitting Step — and expect the tooling to converge.

> **The bar is deliberately high**
>
> From the `/viva-viz` skill: *"A bare line of one observable vs time almost never clears that bar."* Push for interactive Plotly — hover, toggleable legend, sliders — and let the form fit the question: phase portraits, Sankey, heatmap, violin, sunburst. Propose the figure the data deserves; don't just fulfill a request for a line chart.

Three render paths

How a visualization gets its data is the top cause of "my viz renders empty." There are three disjoint paths:

Path	Inputs	Use when
A — inline composite Step	wired ports, per tick	you want a live figure that maintains its own history during the run
B — auto-render from the run store	a typed wire dispatched from `runs.db`	the common default: `'float'` → last scalar, `'list[float]'` → full series, `'list[list[float]]'` → list-of-runs
C — direct store read	empty `inputs()`, you open the store yourself	nested coordinate arrays the typed wire would truncate (e.g. 3D structural viewers)

Post-run analyses and the Analysis tab

An **analysis** is the sibling of a visualization with one load-bearing difference: it **derives an artifact but renders no verdict**. An `AnalysisStep` produces a PNG, JSON, CSV, or Markdown output and surfaces under **Evidence › Analyses**; a curated figure surfaces under **Evidence › Visualizations**. Keeping the two apart is what stops an analysis from quietly implying a pass/fail it never computed.

After a study run completes, the workbench automatically runs every Analysis Step declared in the study's `analyses:` list over the run's emitter output, mirroring how the `visualizations:` list triggers HTML rendering. Each entry names a registered analysis class plus optional `params`; outputs land under the run's directory and their paths come back in the run response. The post-run analysis hook reads the **Parquet** emitter output, so a SQLite-only run skips analyses — a real gotcha worth remembering.

> ⚠ **Two things share the name 'Analysis tab'**
>
> The rail's **Analyses** page is a gallery of *saved, special interactive viewers* (embedded 3D structural scenes, a PTools launcher) — not the catalog of every analysis class. The class catalog lives under **Registry** → **Discovered** → **Visualizations / Analyses**. The docs and the shipped rail labels have drifted here; trust the running UI.

Runs and the run store

Not every question needs authored code. Every run in the workspace is catalogued on the **Runs** rail page — one normalized index that unifies runs across emitter backends (SQLite `runs.db` and Zarr `XArrayEmitter` stores), so an externally-produced run appears there once it is registered. From a run you open its per-study **Results** view, which reads the emitter store — scalar, vector, and bulk observables — without your writing any code.

> ⚠ **Accuracy note — the standalone Data Explorer was removed**
>
> Earlier builds shipped a no-code **Data Explorer** panel with four whole-cell-specific views (Timeseries, run-vs-run Scatter, a Voronoi Allocation treemap, and an Escher Flux map keyed to the *e. coli core* map). It was removed in vivarium-workbench PR #912 (2026-08-20) as too domain-specific for a general workbench — the `/api/explorer/*` routes and the standalone page are gone, and the per-run "view run" button now opens the study Results view instead. The *generic* run-data readers it was built on (`lib/explorer_data.py` — series / observables / vector / flux over parquet / zarr / sqlite) survive and back the Results view and default visualizations. The workbench's own `docs/data-explorer.md` still describes the removed panel; trust the code at HEAD.

Report cards

A **report card** is a Step that reads a completed run and produces **pass/fail outcomes keyed by test**, rendered as a category scorecard. It is the point in the pipeline where evidence becomes a verdict — and the whole design turns on one rule.

> A study's conclusion is **computed from its evidence, not asserted.** You do not write `status: pass`. The verdict is derived from the latest run's measured outcomes — which is what stops a study from *claiming* a result it never produced.

Two things drive that derivation, and neither is a human typing a verdict:

- **Auto-evaluation on run completion.** When a run finishes, the completion path fills that run's `runs[].outcomes` with normalized PASS / FAIL / PARTIAL / SKIP verdicts — but only for tests with no human-authored verdict, so it never clobbers an expert's call.
- **On-demand grading.** A fast "Run tests" path (`POST /api/study-grade`) grades the declared behavior tests against the latest completed run without re-simulating, and refreshes the card.

The compiled card is written to `<study>/viz/report_card/<card>.html` with a structured verdict at `<card>.verdict.json`. That is the *same* artifact the live **Tests tab** shows and the published static report reuses — so a reviewer reading the offline bundle sees exactly the report cards a reviewer sees in the workbench.

The category scorecard

A report card groups its axes by **category** — for example Physiology, Composition, Ribosomes, Exchange fluxes, and Gene expression — and renders each axis as an *axis / value / verdict* row. Those category names are data, not a code-enforced schema: the renderer groups by whatever category slug the axes carry, so a workspace picks its own. Verdicts, by contrast, normalize to a fixed four-value vocabulary (with pass/warn/fail/skip aliased in):

Verdict	Glyph	Meaning
`within_tol`	✓	measured value is within the acceptance band
`drift`	≈	partial / warned — moving, not yet failing
`mismatch`	✗	outside the band
`ungraded`	–	no run outcome yet

Physiology

Axis	Value	Verdict
Doubling time	44 min	✓ within tolerance
Growth rate (μ)	0.94 h^{-1}	✓ within tolerance
Cell mass at division	1.06× ref	✗ above band

Composition

Axis	Value	Verdict
Protein mass fraction	0.55	✓ within tolerance
RNA/protein ratio	0.41	✗ below band

Illustrative only — the numbers, axes, and ✓/✗ verdicts on a real card are computed from a run's outcomes against cited acceptance bands, never hand-set.

Every ✓/✗ traces back to a **behavior test**: a machine-checkable spec that names *how to measure* a result from the run and *what passing means* (an acceptance band, ideally cited to literature). The measurement, the band, and the verdict are one declaration that drives both execution and the rendered pill — which is why the card cannot lie about what happened. The grammar of those tests, the acceptance bands behind the glyphs, and how verdicts roll up into a study-level and investigation-level judgment are the subject of Rigor & the evidence engine.

The published-report aesthetic

The end product a reader receives is a **self-contained, interactive, single-file HTML bundle** — the whole investigation, its verdict DAG, its findings, and its embedded interactive figures in one file that opens with no server. Because the static bundle and the live dashboard render from the *same* declared evidence (figures, report cards, verdicts), the published report and the working session look identical. That is not cosmetic: it is the guarantee that what you send a reviewer is exactly what you graded.

These reports are what agents produce and what the `/viva-report` skill regenerates — deterministically, from the fields the science wrote, with no AI in the rendering path.

Next: Investigations

investigation study

Investigations

A single study answers one question. An **investigation** is the argument several studies build together — a named collection of studies under one research question, wired into a gated DAG that accumulates into a defensible claim.

> *A Composite is the runnable object; a Study is the reason you run it; an Investigation is the argument several studies build together.*

Read the collection top-down and it's an argument; read it bottom-up and it's a pile of simulations. The investigation is the layer that makes the difference — the place where a dozen questions become evidence the field can re-run.

On this page

Assumes Studies. · **You'll learn** the investigation ≡ branch ≡ worktree identity, how the gated DAG accumulates a claim, and why reproducibility follows the branch.

Investigation ≡ branch ≡ worktree

An investigation is not just a folder of studies. It is a **1:1:1 identity**: the slug is also a git branch name and a git worktree directory.

```
investigations/<slug>/investigation.yaml   # the collection + its narrative
spine
investigations/<slug>/studies/<slug>/       # member studies, nested
investigations/<slug>/reports/              # the rendered per-investigation
report
```

The `/viva-investigation` skill makes this concrete:

- `new` writes `investigation.yaml`, creates the branch, and commits — no push.
- `open` materializes a **worktree** at `<ws>/.pbg/worktrees/<slug>/` with its own dashboard server (its own `.pbg/composite-runs.db`, its own ports).

This is why parallel agents never trample each other: one investigation, one branch, one worktree, one server. An investigation is the unit of a branch and a draft PR — the natural boundary at which a line of work is proposed, reviewed, and merged. (Merges are always a human call; the tooling never auto-merges.)

> **Reproducibility follows the branch**
>
> Because an investigation *is* a branch, its history is its audit trail. Every workbench write is a git commit in the worktree, so the verdict, the readiness score, and the open debts are all diff-able. Checkout is archive plus current state.

The `investigation.yaml` narrative spine

`investigation.yaml` carries `schema_version: 1` (minimal) or `2` (the current narrative spine). The minimal form is the skeleton every investigation has:

```yaml
schema_version: 1
name: dnaa-replication-timing
title: "DnaA controls replication initiation timing"
status: in-progress
question: "Does the DnaA-ATP fraction gate replication initiation in the model?"
hypothesis: "Initiation fires when DnaA-ATP crosses a threshold."
studies:
  - dnaa-atp-fraction-baseline
  - dnaa-titration
  - initiation-timing-readout
acceptance_criteria:
  - {study: dnaa-titration, behavior: atp-fraction-in-range}
```

The `studies:` list is the membership (the loader also accepts `members:`). `acceptance_criteria` are `{study, behavior}` pairs that link an investigation-level criterion to a member study's named behavior test — the thread that later lets the report show which criteria are actually covered by evidence.

Schema v2 adds a **nine-section spine** that a report renders automatically:

Section	What it holds
`executive`	the state-first opening: `what_is_this`, `verdict`, `verdict_status`, `verdict_detail`, `decisions_needed[]`
`scientific_argument`	the reasoning chain the studies build
`biological_story`	the mechanism in prose
`lead` · `at_a_glance`	the owner; `[{study, role}]` one-line member roles
`how_to_read` · `glossary`	reader orientation
`guidelines`	investigation-wide rules
`inputs`	owned datasets, references, expert docs

`verdict_status` is a small enum — `planning · in-progress · passed · complete · blocked · failed` — and, like everything else that looks like a conclusion, it is governed by the derive-on-read discipline: authored intent and computed evidence live in parallel fields so the machine never silently overwrites the human.

Prerequisite gates and the computed DAG

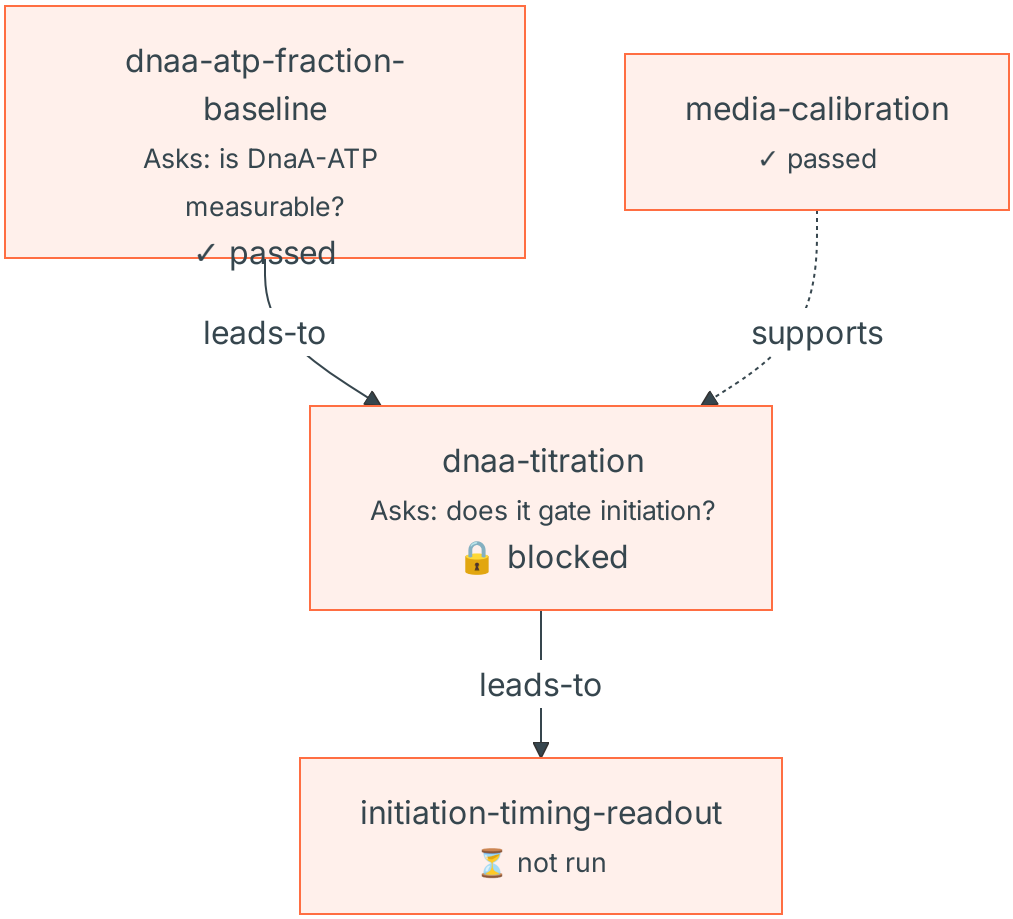
An investigation stores **no edges**. The dependency DAG is **computed at render time** from each member study's `pipeline_gate.prerequisites`. A prerequisite is a bare slug or a small record:

```yaml
# in a member study.yaml
pipeline_gate:
  prerequisites:
    - {study: dnaa-atp-fraction-baseline, condition: tests-passed, relation:
leads-to}
    - {study: media-calibration, condition: ran, relation: supports}
```

The `relation` carries **discourse semantics** — it says what *kind* of argumentative link the edge is, and the graph renders it accordingly:

Relation	Edge	Reading
`leads-to`	solid	this result sets up the next question
`supports`	solid	this result is evidence for another
`regulatory`	dashed	a modulating dependency
`refutes`	dashed	this result argues against another

(The `/viva-study` convention adds a `refines` relation — a finer-grained realization — also drawn dashed.) Each node in the rendered graph reads as an **Ask** (its `question`) → **Finds** (its `claim` or top finding) → a **Confidence** badge, so the whole investigation is legible as an argument at a glance.



A member whose prerequisite has not passed renders 🔒 **blocked** — and here is the subtle part.

Investigation-as-composite

Since August 2026 (vivarium-workbench PR #715, merged 2026-08-03 on `main`), an investigation is not merely *described* by a DAG — it is **compiled into a process-bigraph composite**. Each member study becomes a `StudyStep` node — a real `process_bigraph.Step` wrapping that study — and each `pipeline_gate.prerequisites` edge becomes **store wiring**: a prerequisite is expressed as an input wire, so the engine's own scheduler orders the StudySteps by data dependency. There is no separate gate evaluator; gating *is* process-bigraph's ordinary producer/consumer triggering.

This reframes what a gate means:

> Gating a study is **not** "deciding not to run it." An investigation is an investigation *template* — one open site per member. Admitting a member fills its site; a member whose prerequisite is unmet is left **open and pruned** at compile time. No scheduler ever decides "don't run this node" — the node simply isn't in the ground document.

That is the same **one object / one operation / one law** collapse from the stack: a study is a template with its model site filled; an investigation is a template with one site per study, filled by membership; gating and template-binding are the same mechanism. Running "one study" or "continue from here without rerunning the expensive upstream" is then just `trigger(document, target)` doing pull-or-compute over the results sites — an already-satisfied prerequisite is read from its content-addressed cache rather than recomputed.

> ⚠ **Accuracy note — two run paths coexist**
>
> Investigation-as-composite has shipped, but it runs alongside an older orchestrator. On that older path, the study-to-study DAG is **advisory**: the workbench computes `blocked` / `blocked_by` for the UI, but no run endpoint enforces study-to-study gating — the only runtime gate there is a study's own `conditions.model_settings` marked `gate: required-before-run` with an unset value. The compile-and-prune semantics above are the investigation-composite path (`build_investigation_composite`, `StudyStep`, `templates.trigger`). Which enforcement you get depends on how the investigation is run; verify against the current code before relying on a gate to *block* rather than merely *flag*.

The roll-up

The payoff of grouping studies is that the investigation computes a summary across all of them — none of it hand-written:

Verdict

The rolled-up conclusion state — computed from members' outcomes, not asserted. `roll_up_acceptance` scores the investigation's `acceptance_criteria` against each member's behavior outcomes: any failing criterion → `failing`, else any still running → `in-progress`, else any caveat → `passing-with-caveats`, else `passing`. It writes only `executive.computed_verdict_status`; the parallel authored `verdict_status` is never overwritten. (A single study rolls up its *own* tests separately, via `roll_up_verdict` → passed / failed / needs-calibration / blocked.)

Readiness

A **count of open gaps** — the "6 gaps" style signal from the report linter, telling you exactly which fields are missing or under-supported *before* the verdict is trustworthy. A missing field is itself the feedback.

Open debts

The open epistemic debts across members — the questions owed, the `decisions_needed`, the unresolved feedback. Surfaced so an unproven claim can't masquerade as settled.

Two roll-up behaviors are worth internalizing. First, the summary is designed so that a member lagging in an early phase — one study stuck in Design — holds the whole investigation back rather than letting a single finished study over-report progress. (Verify the exact lifecycle-rollup rule against the current index code; the guarantee that matters is that the summary never *over*-reports.) Second, because every member's verdict is derived from its latest run and the rigor scorecard is a deterministic `ok / warn / gap` per dimension, hardening an investigation is a *transformation an agent can apply and a human can verify* — not a matter of taste.

> *Because the investigation is typed data, hardening it — filling gaps, citing bands, tightening a claim — is a transformation an agent can apply and a human can verify, not a vibe.*

That deterministic evidence engine — behavior tests, acceptance bands, verdict roll-up, and the rigor scorecard — is the subject of the next chapter.

Next: Rigor & the evidence engine

provenance report-card rigor

Rigor & the evidence engine

A study's conclusion is **computed from its evidence, not asserted**. This chapter is about the machinery that makes that sentence true — the deterministic pipeline that reads a run, measures it, rolls the measurements up into a verdict, and writes the whole chain back into the study's own files so a human can audit every hop.

On this page

Assumes Studies, Investigations. · **You'll learn** the evidence spine, the parallel-slot convention, and why the verdict is computed rather than asserted.

The big idea: code where prose used to be

An investigation is a structured research effort — a shared question, a set of studies, each with baselines, variants, runs, acceptance criteria, findings, and expert feedback. Traditionally, the connective tissue between "here is a run" and "here is what it means" is **prose**: an agent or a scientist writes a paragraph asserting that the model passed. Prose is where results quietly drift away from the evidence that is supposed to back them.

The **evidence spine** (the *investigation spine*) replaces that prose tissue with **code**. Information flows from sources to conclusions through deterministic Python functions that read structured data, compute results, and write them back into the project's YAML files. Nothing in this path is an LLM: it is ordinary, testable, reproducible Python. The AI's job is to *author* the inputs — the question, the tests, the mechanism prose — while the spine *derives* everything downstream.

> *You never hand-write `status: pass`. The verdict is a function of the latest run's measured outcomes — which is exactly what stops a study from claiming a result it never produced.*

The parallel-slot convention

The spine's central trick is deceptively simple. For every place a human (or an AI acting as author) records **intent**, there is a **parallel slot** the code fills with **derived evidence**. The two never share a field. Authored intent is preserved verbatim; computed evidence lands beside it; and the dashboard compares them.

What it is	Authored slot (human intent)	Computed slot (code-derived)
Per-run test results	`runs[].outcomes`	`runs[].computed_outcomes`
Study conclusion	`executive.verdict`	`.computed_acceptance`
Gate state	`pipeline_gate.gate_status`	`.gate_evaluator`
A finding	`finding.statement` / `.summary`	`finding.evidence` / `.expected` / `.provenance`

When the computed slot disagrees with the authored one, the reconciler raises a `diverges_from_authored` flag — and *that flag is the dashboard headline*. A study whose author wrote "passing" but whose code-computed acceptance says otherwise does not render as passing; it renders as **diverged**, in red, with both values shown. Drift becomes impossible to hide because the disagreement is a first-class, diffable fact.

Why the split earns its keep

Authored and computed values live in the same file but in different fields, so a single `git diff` shows you *both* what a claim asserts and whether the evidence still supports it. Auditability, drift detection, and self-documenting studies all fall out of this one convention.

Where these live in code

All four coded slots are real fields in `viva_superpowers`, not just design vocabulary: `runs[].computed_outcomes` (written by `study_evaluator` / `study_outcomes`), `executive.computed_acceptance` (`investigation_status`), and `pipeline_gate.gate_evaluator` (`study_verdict.write_gate_evaluator`) — the last two each carrying the `diverges_from_authored` flag. The `study_outcomes.sync()` entry point drives the reconciliation. Each write is a comment-preserving ruamel round-trip that only ever fills the coded slot and never touches the authored one beside it.

Compute → Show → Act

The spine is layered into three planes. The bottom plane computes; the middle plane renders what was computed; the top plane turns the rendered state into the next scientific action.

1 · Compute

The eight-stage spine. Deterministic Python reads sources and runs, measures outcomes, rolls them into verdicts and findings, and reconciles authored ↔ computed with one `study_outcomes.sync()`.

2 · Show

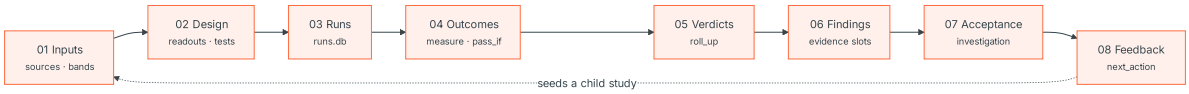
Spine presentation. Code-vs-authored chips, `diverges_from_authored` badges, per-readout validation marks, and a verdict-annotated study graph — all rendered by the **AI-free** Workbench.

3 · Act

The active layer — **Propagate · Navigate · Guide**. Expert feedback becomes a tracked action, which writes a finding's `next_action`, which seeds a child study. The loop closes on itself.

The eight-stage spine

Every study's evidence flows through the same ordered stages. Each stage reads the one before it and writes a different part of the study file.



The reflexive loop at the end — feedback seeding the next study — is what makes the spine an *engine* rather than a report format: an investigation propagates, shows, and acts on its own knowledge.

The spine gates

Between stages sit **gates**: each is a deterministic verifier with a small, fixed result vocabulary. A gate never guesses; when it cannot resolve something it says so explicitly, in a bucket a human can act on.

Gate	Question it answers	Result vocabulary
Observable	Does this readout resolve to a real emission in the model's structure?	`ok` · `unresolved` · `not_in_structure` · `aspirational`
Outcome	Did the run clear the test's bar?	`PASS` · `FAIL` · `PARTIAL` · `BLOCKED`
Citation	Is this acceptance band backed by a literature source?	cited · uncited
Parameter drift	Did the run honor the study's enforced parameters?	in-band · drifted
Study	Do this study's prerequisites and outcomes let it conclude?	passed · failed · needs_calibration · blocked
Investigation	Do the member studies' verdicts satisfy the acceptance criteria?	met · partial · unmet
Feedback	Has each expert item been resolved into an action?	open · actioned · closed

These compose into a single **gate trail** — the audit path from a raw source all the way to a resolved expert note:

```
source → band → readout validation → run → outcome → study verdict → finding → investigation acceptance → feedback status
```

The **observable gate** deserves special emphasis because it is the *never-fabricate guard*: a readout marked `not_in_structure` means the study declared an observable the composite cannot actually emit. The spine refuses to invent a number for it — it buckets it as unresolved and tells you a human (or a model change) is needed.

The three rules

Every write the spine makes obeys three rules. They are the ethical spine of the evidence spine, and they recur across every design document.

Never clobber the human

Authored intent and machine evidence live in **parallel fields**. Code fills the *absent* computed slot; it never overwrites what a person wrote. Your `verdict` stays exactly as you typed it — beside, not under, the computed one.

Never guess

When something cannot be resolved, the spine **buckets it** (`code` · `agent` · `needs_rerun`) rather than fabricating a verdict. A closed, `eval`-free DSL means a measurement either computes honestly or reports that it could not.

Never destroy provenance

Every write is **comment-preserving** (ruamel round-trip) and lands as a **git commit**. Verdict, readiness, and open debts are reproducible, auditable, and diffable. The history *is* the evidence trail.

Behavior tests as machine-checkable specs

The unit that drives all of this is the **behavior test**: not prose, but a spec that names *how to measure* a result from a run and *what passing means*. Because the same definition drives both execution and the dashboard's pass/fail rendering, a test is required to have four properties:

- **Executable** — a `measure` (a path into the run's structure, a reduction, a window) that a deterministic reader can evaluate against `runs.db`, with no human in the loop.
- **Honest** — a `gate_class` that distinguishes an *acceptance criterion* (a directional prior stated before the run) from a *regression pin* (a threshold set after seeing the data), so a study cannot quietly tune the model to pass a bar it drew after the fact.
- **Traceable** — `requires_simulation` and `cites` bind the verdict to a *specific run* and to the *literature*, so the number and its justification travel together.
- **Renderable** — one structured object drives execution *and* the pass/fail pill, so what you read on the dashboard is the same fact the grader computed.

A worked example: the DnaA-ATP fraction

Here is a real behavior test from the `dnaa-replication` line of work. It asks whether the fraction of DnaA bound as DnaA-ATP stays inside its biologically expected band, **in every generation**, over a multi-generation run.

```yaml
behavior_tests:
  - name: dnaa_atp_fraction_in_band
    classification: primary
    gate_class: acceptance_criterion
    measure:
      kind: generation_average
      path: "MONOMER0-160[c] / (PD03831[c] + MONOMER0-160[c] + MONOMER0-
4565[c])"
    pass_if:
      op: in_range_every_generation
      low: 0.2
      high: 0.5
    requires_simulation: baseline
    cites: [Boesen2024]
```

When the report card runs, the spine resolves each `[c]` token to a real store path, evaluates the ratio with a safe AST resolver (no `eval`), averages it per generation, and checks the band. It does **not** write `pass: true`; it writes a structured, per-generation measurement:

```
computed_outcomes:
  dnaa_atp_fraction_in_band:
    evaluated_by: code
    result: PASS
    operator: generation_average/in_range_every_generation
    measured_value: { 1: 0.499, 2: 0.421, 3: 0.388, 4: 0.352, 5: 0.310, 6:
0.271, 7: 0.225 }
```

The verdict is now *derived* from those seven numbers. If generation 7 had dipped to `0.18`, `result` would flip to `FAIL` on its own — and if the authored `executive.verdict` still said "passing," the `diverges_from_authored` flag would fire. The measurement chain that made this possible runs `structure → readout → self-describing store → RunReader → measure → verdict`, with the run reader living in the sibling emitters package so the dashboard never has to.

> **A behavior test is a machine-checkable spec, not prose.**
>
> Two studies that cite the same band and measure the same path will grade the same way, forever, on any machine that can replay the run.

Hardening a study

A study that *looks* done is not the same as a study whose conclusions survive scrutiny. **Hardening** is the transformation that closes the gap between the two — and because a study is typed data, hardening is a transformation an agent can apply and a human can verify, not a matter of taste.

Three moves carry most of the work — one standalone skill plus two of its sibling subcommands:

Move	What it hardens
`/viva-harden-investigation`	Finds the **one load-bearing gap** that most weakens the headline claim — an unbacked scaffold, a knife-edge single-seed pass, an overclaimed verdict, a failing gate with a real divergence — and closes *that one* the right way for its kind, rather than pouring generic rigor onto an un-diagnosed failure.
`/viva-tests cite-bands`	Surfaces candidate evidence from expert PDFs for **uncited acceptance bands** and writes structured `cites` / `calibration_anchor` provenance into the study — so a threshold traces to a paper instead of being a magic number. Every write goes through the sanctioned `POST /api/band-provenance` path; the skill never fabricates a threshold.
`/viva-harden-investigation biology-forward`	Runs the deterministic populate step first (`populate_finding_observations` fills `evidence.observed`, `expected.range`, `divergence_factor`, `provenance.run_ids` from the computed outcomes and bands), **then** guides the author to write the biological interpretation over that scaffold. A `divergence_factor > 0` forces the status to `partial` or `contradicts` — never `confirms`. Numbers first, mechanism prose second.

Skill surface consolidated (#281)

At the current package HEAD the `/viva-*` surface was consolidated (21 → 17 skills), so band-citing is now the `cite-bands` subcommand of `/viva-tests` and biology-forward authoring is a mode of `/viva-harden-investigation`. Older installed plugin builds may still expose `/viva-cite-bands` and `/viva-biology-forward` as standalone skills; the behavior is the same either way.

Warning

Hardening never means "sprinkle more seeds everywhere." A failing report-card gate with a real divergence must be **root-caused before it is patched** — and a verdict of *real and understood* (no bug found) is itself a complete hardening. The point is to explain the signal, not to bury it under undifferentiated rigor.

Authored vs computed: the whole picture

Pulling it together, here is who writes what across a study's spine. The left column is where human (or AI-as-author) judgment lives; the right is what the deterministic engine derives.

Stage	You (or an AI) author...	The spine computes...
Design	the question, readouts, behavior tests, acceptance bands	readout resolution status (observable gate)
Runs	which composite to run, with what perturbations	`runs.db` trajectories + run provenance
Outcomes	—	`computed_outcomes` per test (measure → `pass_if`)
Verdicts	your intended `executive.verdict`	`computed_acceptance` + `diverges_from_authored`
Findings	`statement`, `summary`, `explanation`, `status`	`evidence`, `expected.range`, `divergence_factor`, `provenance`
Acceptance	the investigation's acceptance criteria	which criteria are met / partial / unmet
Feedback	the expert note and the decision it demands	the tracked action → the child study it seeds

> You can re-run the science without rewriting the argument — and audit the argument without re-reading the code.

Next: Working with AI agents

agents api workbench

Working with AI agents

The agentic spine is *what reasons* — but it reasons through a tool that contains no AI at all. This chapter is about that seam: how the `/viva-*` skills drive the Workbench over a plain HTTP contract, and how the work divides between an AI agent doing high-throughput construction and a human curator owning the calls that matter.

Same artifact, two kinds of author.

The agentic closed loop: hypothesis, prediction, workflow contract, simulation, evidence package, validation, and model update, over a process-bigraph execution substrate with human approval at each stage.

The agentic loop: AI agents assemble and run the investigation graph — hypothesis → prediction → workflow → simulation → evidence → validation → model update — over a process-bigraph execution substrate, with a human approving the stages that matter.

On this page

Assumes Workspaces & the Workbench, Rigor & the evidence engine. · **You'll learn** the AI-free-tool / swappable-plugin split, how the `/viva-*` skills drive the Workbench over plain HTTP, and how work divides between agent and human curator.

The split: AI-free tool, swappable AI plugin

The single most important architectural decision for agents is that the AI and the tool are **two separate layers, and they stay separate**:

> **The AI layer — `viva-superpowers`.** The `/viva-*` Claude Code skills. All AI capability in the ecosystem lives here. The skills are thin clients: shell, `curl`, and Python-helper calls against the Workbench's HTTP API. Swap this layer for a different model or a different agent and nothing below it changes.

> **The tool layer — `vivarium-workbench`.** The dashboard server, its data, its evidence rendering. Pure Python, **no AI dependencies**. It runs composites, renders the study and investigation UI, and commits every change to git. It behaves identically whether an AI, a script, or a human at a keyboard is driving it.

Why go to this trouble? Two payoffs, and they are the whole reason the platform can be trusted:

- **Auditability.** The evidence you read on the dashboard was rendered deterministically from declared fields by code with no model in the loop. There is no prompt that could have talked it into a rosier verdict. The evidence spine computes; the AI only authors the inputs.
- **Replaceable AI.** Because every capability the AI has is expressed as an HTTP call any client could make, the AI is a *plugin*, not a dependency. The tool outlives any particular model.

All AI capability is packaged as the `viva-superpowers` plugin — a set of `viva-*` skills that call the Workbench's HTTP API. **This keeps the tool auditable and the AI swappable.**

The agent access contract

Any agent — a `/viva-*` skill, a script, a different model entirely — drives the Workbench through the same small contract. Follow it and your agent behaves; break it and you get the classic failures (a hardcoded port that moved, a "success" that was really a failed run returning HTTP 200).

1 · Resolve the base URL — never hardcode it. On start, the server writes a small JSON blob to `.pbg/server/server-info` whose `url` field is the live base URL (alongside `port`, `host`, `pid`). That file is how *everything*, agents included, discovers the server. Read the `url` key; never assume a port.

```
BASE=$(python3 -c "import json; print(json.load(open('.pbg/server/server-info'))
['url'])")  # e.g. http://127.0.0.1:8765
curl -s "$BASE/api/workspace-manifest" | head
```

2 · Orient in one call, then two. Before editing anything, get situational awareness. `GET /api/workspace-manifest` returns the workspace, composites, studies, registry, health, and installed skills in a single snapshot — start there. `GET /api/linkage-index` returns the deterministic cross-reference graph: what links to what, which studies cite a source, which findings measure an observable.

3 · Runs are asynchronous — check the result, not the status. A run returns a `run_id` and comes back later. Poll its status endpoint until it is done, then read the run's **result field**. A *failed run can still return HTTP 200* — the durable truth is in `studies/<slug>/runs.db`, not in the HTTP envelope. Confusing "the request succeeded" with "the run passed" is the most common agent mistake.

4 · Every write is a git commit. There is no separate "save." Each mutating call the tool commits to git, so every action an agent takes has an audit trail and is reversible. This is what makes high-throughput, agent-driven construction safe: nothing an agent does is invisible or unrecoverable.

The golden rules, in one line

Read `.pbg/server/server-info` for the URL · orient via `/api/workspace-manifest` first · trust `/openapi.json` for shapes · runs are async (poll, and check the *result*, not just HTTP 200) · every write is a git commit. See the HTTP API reference.

The authoring arc

Most agent sessions walk the same arc, from an empty workspace to a closed investigation with a report. Each step is a skill; each skill is a client over the contract above.

Step	Skill	What the agent does
Scaffold & serve	`/viva-workspace`, `/viva-workbench`	Create the workspace (`workspace.yaml` + a `viva_<pkg>/` package), start the dashboard server.
Catalog	`/viva-catalog`	List, install, or uninstall the workspace's module catalog — pull in the processes you need.
Wrap	`/viva-expert`	Wrap a real upstream simulator as a typed Process (or compose several). The default is to bridge the *genuine* tool, never a stub.
Compose	—	Assemble the wrapped processes into a composite — the only object the engine runs.
Ask	`/viva-study`	Wrap one question, one emit-contract, and one pass/fail bar around the composite.
Run	`/viva-run`	Execute a composite for N steps; poll to completion; read the emitted observables.
Render	`/viva-viz`, `/viva-report`	Generate interactive figures and regenerate the reviewer-ready report.
Argue	`/viva-investigation`	Group studies under one research question into a gated DAG; close it into a report and a PR.

The full set is in the skill catalog. The arc is a loop, not a line: a finding's `next_action` seeds the next study, and the agent rides that loop at every hop — proposing studies, running composites, hardening findings, reading verdicts to choose what to ask next — while every step stays legible to a human on the study page.

Curator and agent: the division of labor

The investigation graph is a **shared surface** that two very different kinds of author edit. Getting the boundary right is what makes high-throughput AI construction *trustworthy* rather than merely fast.

The agent does the volume

High-throughput **construction and analysis**: building studies, wrapping simulators, running and sweeping composites, drafting visualizations, populating finding scaffolds, maintaining the graph. Agents work **in parallel, in isolated worktrees** — one investigation per branch per worktree — so concurrent sessions never trample each other's runtime state.

The curator owns the judgment

The **irreversible, framing decisions**: what question is worth asking, whether a verdict is justified, when to merge, whether a correction should propagate upstream. These are matters of scientific taste, and they stay with a human.

The co-authoring loop, in five beats: the **curator frames** → the **agent implements** → the **spine executes and the grader scores** (PASS / PARTIAL / FAIL, deterministically) → the **agent analyzes** → the **curator reviews, sets the verdict, and decides the merge**; corrections then propagate upstream.

Two responsibilities the curator must never delegate, because the structure exists specifically to guard against them:

> ⚠ **Guard against tune-to-pass and over-claiming**
>
> - **Tune-to-pass.** An agent optimizing for a green pill will happily draw the acceptance band *after* seeing the data. The `gate_class` split (an `acceptance_criterion` — a directional prior stated before the run — versus a `regression_pin` set afterward) and pre-registered thresholds exist to catch this, but the curator is the one who verifies a bar was set honestly, before the run.
> - **Over-claiming.** A confident verdict on thin evidence is the failure the `diverges_from_authored` flag and `/viva-harden-investigation` are built to surface. The curator reconciles the claim to what the evidence actually supports.

Never auto-merge. A merge is a scientific commitment; the human always approves it. AI augments the investigation — it never replaces scientific accountability.

The reason the boundary can be this clean is everything in the two preceding chapters: the tool is AI-free, so what the agent produces is graded by code; the evidence is computed, not asserted, so a curator reviewing an agent's study is reviewing *facts the spine derived*, not prose the agent wrote. Structure is what makes an agent-assembled model trustworthy rather than plausible-looking.

Next: Skill reference

PART 4

Reference

Look-up material — the exhaustive detail behind the earlier parts. Come here when you need the exact skill, route, file shape, or install step, plus a full worked example and a glossary of every load-bearing term.

Skill reference

The full `/viva-*` catalog — every skill, what it does, and which API or helper it wraps.

HTTP API reference

The Workbench request surface, organized along the Investigations → Studies → Runs spine.

On-disk schema reference

The field guide to the YAML and JSON shapes a workspace stores and the Workbench commits.

Install & deploy

Getting a workspace and the Workbench running — laptop, container, and read-only site.

A worked end-to-end example

One question followed from an empty directory to a graded study and the next study it seeds.

Glossary

Every load-bearing term in one place, with its formal anchor and its legacy synonyms.

Module catalog

Every installable Vivarium module — searchable and filterable by capability tag.

Suggested path

Reference is for look-up, not linear reading — jump to the entry you need. If you want it to cohere, follow **A worked end-to-end example** and open the other entries as it links to them.

agents reference skills

Skill reference

The `/viva-*` skills are the ecosystem's AI layer — a Claude Code plugin (`viva-superpowers`) whose commands author and drive everything else. They are **thin clients**: shell + `curl` + calls into the `viva_superpowers` Python helpers and the Workbench HTTP API. Almost nothing here does useful work on its own; the Workbench is the backend.

This chapter is the catalog — every skill, what it does, and which API or helper it wraps. For the request surface those skills call, cross over to the HTTP API reference; for how an agent drives them end to end, see Working with AI agents.

On this page

What's here the full `/viva-*` skill catalog — every command, what it does, and which HTTP endpoint or `viva_superpowers` helper it wraps. · **See also** the HTTP API reference for the request surface, and Working with AI agents for the loop an agent drives end to end.

Two preconditions every dashboard-touching skill assumes

1. **A workspace** — a directory with a `workspace.yaml` and a `viva_<pkg>/` Python package. Create one with `/viva-workspace`.
2. **A running Workbench server** — started with `/viva-workbench start`. Skills discover its base URL by reading `.pbg/server/server-info`.

Each skill opens with the same preamble: walk up to `workspace.yaml`, read the server URL from `.pbg/server/server-info`, run a version-skew preflight that warns but never fails. A missing precondition fails the skill with an actionable error that names the fix.

`pbg` → `viva` naming

Skills are `/viva-*`; the older `/pbg-*` command names have been dropped (the Python import shim `pbg_superpowers` remains). The runtime control directory is still `.pbg/` and the global registry still `~/.pbg/` — those paths did not change. Some in-repo convention docs still say `pbg` and show `pbg-<tool>` repo names; read them as `viva`. See The stack — a note on names.

Workspace lifecycle

Bootstrap a workspace, then bring the server up.

Command	Purpose	Wraps
`/viva-workspace <name> [--upstream owner/repo] [--in-place]`	Scaffold a research workspace — three modes: **upstream-branch** (clone a model repo, branch a workspace on top), **standalone** (clone `viva-template`), **in-place** (promote an existing git checkout).	`vwb scaffold-workspace` / `catalog-add`; `viva_superpowers.scaffold`
`/viva-workbench start|stop|status|open|restart`	Start / stop / open the interactive dashboard server (the side-rail-tabbed UI). Session-per-tab: one server multiplexes many workspaces, one per browser tab. `open --composite <spec-id>` opens the **Composite Explorer** focused on a spec. Formerly `/pbg-dashboard`.	`vwb serve` / `server-*`; writes `.pbg/server/server-info`, registers `~/.pbg/servers/*.json`
`/viva-init`	One-shot, per-machine: symlink the `/viva-*` skills into `~/.claude/skills/` so Claude can invoke them.	filesystem symlinks (fallback: `/plugin install`)

`/viva-workspace` picks its mode from the flags

Pass `--upstream owner/repo` to branch a workspace on top of an existing model repo; pass nothing to clone the standalone template; pass `--in-place` inside an existing checkout to promote it. Composite-only repos (no `workspace.yaml`) are promoted with `--in-place`.

Wrap & compose

One skill, four modes. `/viva-expert` is the process-bigraph API expert: it wraps a simulator as a process-bigraph `Process` / `Step`, or composes several wrapped simulators together.

Command	Purpose	Wraps
`/viva-expert <tool>`	Wrap a simulator as a Process.	process-bigraph API; sibling-repo scaffold `templates/model/`; `core_compose.py`
`/viva-expert <name> <tool1> <tool2> …`	Compose two or more wrapped simulators into one composite.	same, composite terminus

The default bridges the *real* upstream tool — locate it (PyPI, GitHub, binary), install/build it into the wrapper's venv, run a minimal example, then drive it from the Process's `update()`. It keeps trying when the build is hard and **never silently downgrades** to fake behavior. Producing a mock or a reimplementation is always an explicit opt-in.

Mode	Flag	What lands	`update()` drives
Heavy (default)	*(none)*	A sibling `viva-<tool>/` repo — Process class, tests, README, HTML report, a showcase investigation with studies and interactive viz, a **published read-only workbench**, and a local commit.	the genuine installed tool
Lightweight	`--lightweight` (alias `--in-workspace`)	One file `viva_<slug>/processes/<tool>.py` plus a test, **inside the current workspace**. No sibling repo, no publish, no commit. **Still bridges the real tool.**	the genuine tool
Reproduce	`--reproduce` (alias `--reimplement`)	A clean-room, honestly-labeled `<Tool>ReproductionProcess` (secondary class, never the headline).	a clean-room reimplementation
Mock	`--mock` (alias `--stub`)	A labeled, non-functional `<Tool>MockProcess` (real ports, inert `update()`) for scaffolding/wiring only. The **only** path that emits fake behavior.	nothing real

> ⚠ **`--mock` and `--reproduce` are opt-in and mutually exclusive**
>
> The skill never chooses either on its own. If the real bridge genuinely can't run in the environment, it climbs an escalation ladder (PyPI → source build → pinned older release → the tool's own Docker/conda recipe for hints) and then *asks* which flag you want — it does not decide to fake behavior for you.

Studies & runs

Turn composites into runs, and runs into gated evidence. See Studies for the concepts these commands author.

Command	Purpose	Wraps
`/viva-run <composite-id> [--steps N] [--emit p1,p2]`	Smoke-test a catalog composite directly — run N steps (default 5), report emitted observables. No Study touched. Detached: the POST returns `202 {run_id}`; poll status, then read `.pbg/runs/<run_id>/observables.json`.	`POST /api/composite-test-run`, `GET /api/composite-run/{run_id}/status`
`/viva-study <subcommand> …`	Full Study CRUD across the **Design → Build → Simulate → Evaluate → Decide** lifecycle — baselines, variants, interventions, runs, findings, conclusions on a `study.yaml`.	`/api/study-*` (create, run-baseline/-variant, grade); `study_io` / `study_status` / `study_evaluator`
`/viva-investigation <subcommand> …`	Manage Investigations — named collections of Studies under one research question, with a cross-study DAG. Subcommands: `new`, `open [--share-artifacts]`, `list`, `add-study`, `remove-study`, `set-overview`, `set-status`, `scaffold-from-plan <plan.pdf>`, `run [--keep-going]`, `close`. An investigation is a git branch and a worktree.	`POST /api/investigation-create`, `/api/investigation-summaries`, `/api/iset-close`
`/viva-viz <viz-name> '<description>'`	Generate a v2 decorated-function `Visualization` into `viva_<pkg>/visualizations/` from a natural-language description. Pushes for the most interactive, informative figure the data supports.	`process_bigraph.visualization.as_visualization`; a disk handoff — reads `.pbg/viz-requests/<name>.md` (the dashboard writes the request via `POST /api/visualization-generate`; there is no `/api/study-viz-add`)
`/viva-report [model | --`	Regenerate the dashboard + per-investigation reports. Runs a reviewer-	`report.py`, `report_linter.py`;

Command	Purpose	Wraps
`all | --audit | --lint | --force]`	readiness audit (**Pass A** — verdict↔chart drift, stale framings, required new-viz proposals) then a structural lint (**Pass B**) then renders. Run it before sending a report out. Idempotent.	renders `reports/index.html`

`/viva-study` subcommands, by phase

- **Design** — `new <composite-id>`, `fill-overview`, `set-objective`, `baseline-add` / `baseline-remove`, `variant-add` / `variant-set-params` / `variant-delete`, `intervention-add` / `-update` / `-delete`, `add-literature-anchor`, `add-requirement`.
- **Design → Build gate** — `verify`, `check-observables` (guards against fabricating an observable the composite can't emit), `preview-viz`. *Don't run until both pass.*
- **Simulate** — `run-baseline`, `run-variant`, `run-script`, `refresh-viz`, `clean`.
- **Decide** — `set-conclusion`, `set-verdicts`, `findings`, `propose-followup`, `seed-from-followup`. *No verdict without fresh evidence.*

Navigate & status

Read where you are without running anything. These are pure deterministic queries — no AI, no writes.

Command	Purpose	Wraps
`/viva-catalog [list | install <pkg> | uninstall <pkg>]`	Browse or mutate the workspace module catalog — list installed/available modules, install a curated package, uninstall one. `list` is the default.	`GET /api/workspace-manifest`, `POST /api/catalog-install`, `POST /api/catalog-uninstall`; `workspace_catalog.py`
`/viva-navigate <subcommand>`	Read-only knowledge-graph queries. `status` (workspace/server/git check) · `decisions <inv>` (the ranked "what needs your decision" list) · `ac-gaps <inv>` · `source <bib_key>` · `finding-by-observable <token>` · `dag <inv>` · `observable <token>` · `composite <id>`.	`GET /api/linkage-index`, `GET /api/needs-attention`

`viva-status` and `viva-explore` are not separate skills at this HEAD

They fold into the two skills above and into `viva-workbench`:

You may see	It is actually
`/viva-status`	`/viva-navigate status`
`/viva-explore <spec-id>`	`/viva-workbench open --composite <spec-id>` (opens the Composite Explorer)

An older packaged snapshot shipped these as standalone commands; the current `skills/` directory does not. Prefer the folded-in forms.

Evidence & rigor

Make a study's claims defensible: grade it, cite its bands, write its biology.

Command	Purpose	Wraps
`/viva-tests <author | enrich | run | audit | cite-bands> <study> [name]`	Author, enrich, run, audit, and cite a study's **graded report cards** — the `TestStep`s that compile a run into a pass/fail verdict *and* a signed `margin` (distance-to-pass) plus a cross-iteration diff. Bands over magic numbers. `audit` judges whether the tests are rigorous enough to validate *before* they're locked.	`viva_superpowers.check()` / `TestBuilder` (`test_contract.py`), `test_audit.py`, `band_provenance.py`; `POST /api/study-tests-run`, `/api/study-grade`, `/api/study-test-audit`
`/viva-harden-investigation [slug | biology-forward <study-slug>]`	Make an Investigation or Study rigorous — triage the load-bearing claim↔evidence gap, root-cause failing report-card gates, resolve open `decisions_needed`. The `biology-forward <study>` aspect fills quantitative finding slots (`evidence.observed`, `expected.range`, `divergence_factor`) deterministically, then guides you to author the mechanism prose over that scaffold.	`finding_observations.py`, `rigor.py`, `report_linter.py`; `POST /api/study-findings-populate-observations`, `/api/study-sync-runs`

`viva-cite-bands` and `viva-biology-forward` are subcommands

Like `viva-status` / `viva-explore`, these are not standalone dirs at this HEAD:

You may see	It is actually
`/viva-cite-bands <study>`	`/viva-tests cite-bands <study>`
`/viva-biology-forward <study>`	`/viva-harden-investigation biology-forward <study>`

`cite-bands` surfaces candidate evidence from expert PDFs for uncited acceptance bands and writes structured `cites` / `calibration_anchor` provenance into `study.yaml` — its only sanctioned write path is `POST /api/band-provenance` (comment-preserving), never a direct import.

Also shipped

Beyond the groups above, the plugin ships a few more skills for the autonomous loop, remote compute, and session setup.

Command	Purpose	Wraps
`/viva-model-build <study-slug> [--autonomous]`	Drive the agentic model-building loop: author acceptance-criteria Tests → **audit** their sufficiency → **lock** them → build/run/evaluate → iterate the *model* (never the locked Tests) until the severity gate passes or it gives up honestly. Driver only — it never grades.	`loop_state.py`, `module_sourcing.py`, `study_verdict.py`; state in `.pbg/loop/<study>.json`
`/viva-remote-run <connect | submit | status | fetch>`	Run a workspace composite **remotely on viva-api (GovCloud)** — connect (SSO + SSM tunnel), submit, poll, fetch. The composite declares its own emitter. Framework-generic.	the viva-api remote-run flow (see HTTP API — Remote / GovCloud)
`/viva-benchmark <suite> [--variant-label] [--score-only]`	Score the framework's model-building ability: run a suite of open-ended questions through the autonomous loop, grade each with a reference-free rubric, write a `benchmark_report/v1` you can diff across framework variants.	`benchmark_run.py`, `benchmark_score.py`
`/viva-orient`	Session-start orientation to the `/viva-*` skills, the two preconditions, and a routing table. **Auto-injected** by the SessionStart hook — not a command you run.	the SessionStart hook (`hooks/session-start`)
`/viva-suggest <request-id>`	**Internal** dashboard callback for the Workbench "Suggest" button — drafts a repo name, PR title, or PR body from a request file. `user-invocable: false`; the dashboard prints the exact command when it's needed.	`.pbg/agent-requests/` ↔ `.pbg/agent-responses/` files; `POST /api/suggest`

> ⚠ **Accuracy note — skill count drifts**
>
> The repo's own docs disagree on the total (`README` says 15 in one place, 18 in another; `docs/skills.md` says "15 user-facing"). The `skills/` directory at the verified HEAD holds **18 skill directories**; of those, `viva-orient` is auto-injected and `viva-suggest` is not user-invocable. Treat the directory as authoritative and don't rely on a fixed headline number.

Next: HTTP API reference

agents api reference

HTTP API reference

The Workbench is a **FastAPI app under uvicorn** (`vivarium_workbench/api/app.py`). Every `/viva-*` skill drives it; every mutating call commits to the active git branch in the workspace. The surface is large — at the verified HEAD `api/app.py` registers **253 routes** (122 GET, 122 POST, 7 DELETE, 2 PATCH; 245 unique paths, no WebSocket); the in-repo survey's older headline of "260 routes / 135 POST / 119 GET" is stale, so trust a live recount over any fixed number — so this chapter organizes the ones that matter along the **Investigations → Studies → Runs** spine, as reference tables. It is not the generated spec: for exact request and response shapes, read `GET /openapi.json` (also `/docs`, `/redoc`) on a live server.

For the commands that call these endpoints, cross back to the Skill reference; for the agent's-eye view of the whole loop, see Working with AI agents.

On this page

What's here the Workbench REST surface organized along the Investigations → Studies → Runs spine, as reference tables — plus the generated-spec pointers (`/openapi.json`, `/docs`, `/redoc`) for exact shapes. · **See also** the Skill reference for the commands that call these endpoints, and Working with AI agents.

The agent access contract

- **Base URL** comes from `.pbg/server/server-info` — never hardcode a port.
- **No auth, trusted-localhost.** A same-origin CSRF guard covers every POST/DELETE; requests with **no `Origin` header** (curl, a CLI, an agent) are allowed, and a present `Origin` must match `Host`. There is no token and no Host allowlist.
- **Runs are async.** A run returns a `run_id`; poll status and **check the result field, not just the HTTP status** — a failed run can still return 200.
- **Every write is a git commit** in the workspace, so the whole history is an audit trail.

```
# Orient against a running server — the three calls to start with.
BASE=$(tr -d '[:space:]' < .pbg/server/server-info)
curl -s "$BASE/api/workspace-manifest" | jq .      # one-call situational
snapshot
curl -s "$BASE/api/linkage-index"      | jq .      # the cross-reference graph
curl -s "$BASE/openapi.json"           | jq '.paths | keys'   # exact shapes
```

Orientation

Start here. These read the local workspace and tell an agent where it is.

Endpoint	Method	Purpose
`/api/workspace-manifest`	GET	One-call situational snapshot: workspace, composites, studies, registry, health, skills. **Start here.**
`/api/linkage-index`	GET	Deterministic cross-reference graph (`ac_gating_matrix`, `studies_for_source`, `findings_for_observable`, `study_dag`, `composite_emits` …). Backs `/viva-navigate`.
`/api/needs-attention`	GET	The "decisions needed" scan (uncovered ACs, verdict divergence, open feedback, param drift, stale findings).
`/api/events`	GET (SSE)	Server-sent workspace-state stream; `/api/events/log` is the durable `.pbg/events.jsonl`.
`/api/workspace`, `/api/inputs`, `/api/state`, `/api/ui-config`, `/api/server-version`	GET	Workspace summary, declared inputs, UI config, server version.
`/health`	GET	Service liveness (note: no `/api` prefix).

```
GET /api/workspace-manifest HTTP/1.1
Host: 127.0.0.1:PORT
```


```
{
  "workspace": {"name": "my-model", "package": "viva_mymodel", "branch":
"main"},
  "composites": ["viva_mymodel.composites.baseline"],
  "studies":   [{"slug": "overflow-metabolism", "phase": "Evaluate"}],
  "health":    {"ok": true}
}
```

Workspaces & sources

Switch which workspace or which built simulator a session points at.

Endpoint	Method	Purpose
`/api/workspaces`	GET	List known workspaces + running dashboards.
`/api/workspaces/add`, `/forget`, `/start`, `/stop`, `/cleanup-stale`	POST	Register, drop, start, stop, or GC a workspace server.
`/api/source/manifest`	GET	The provenance manifest `{repo, commit, branch, workspace, lockfile, results, simulator_id}` — the join key for the round-trip pipeline.
`/api/source/builds`, `/api/source/materialization`, `/api/source/remote-health`	GET	Available builds, staging/venv materialization status, remote health.
`/api/source/switch`, `/api/source/switch-build`	POST	Point the session at a repo/ref, or at a built `simulator_id`.
`/api/source/build-remote`, `/api/source/materialize-repo`	POST	Ask viva-api to build a `repo@branch`; stage a repo locally.

Composites

Resolve, inspect, run, and promote composites. The Composite Explorer is built on these; `/viva-run` calls `composite-test-run`.

Endpoint	Method	Purpose
`/api/composites`	GET	The registry — every composite the workspace can import.
`/api/composite-resolve`, `/api/composite-state`, `/api/composite-inner-state`	GET	Resolve an id to a document; read its state tree.
`/api/composite-layout`, `/api/composite-default-view`	GET	Explorer layout + default view for the loom viewer.
`/api/composite-config-translate`, `/api/config-to-composite`	GET/POST	Translate between a config form and a composite document.
`/api/composite-config-persist`	POST	Explorer config upload → parameter becomes an absolute path.
`/api/composite-promote-to-catalog`	POST	Save a configured composite into the workspace catalog.
`/api/composite-test-run`	POST	**Dispatch a detached scratch run** — writes a request JSON, spawns the runner fully detached, returns `202 {run_id, status:"running"}`.
`/api/composite-runs`	GET	List scratch runs.
`/api/composite-run/{run_id}`	GET	One run; sub-paths `/status`, `/state`, `/stop`, `/download`, `/artifact/{name}`.

```
# Fire a 10-step scratch run, then poll it (the /viva-run pattern).
RID=$(curl -s -X POST "$BASE/api/composite-test-run" \
      -H 'Content-Type: application/json' \
      -d '{"composite_id":"viva_mymodel.composites.baseline","steps":10}' | jq -
r .run_id)
curl -s "$BASE/api/composite-run/$RID/status" | jq .   # poll until done, then
read observables
```

Studies

The largest area (**~54 routes** at HEAD). A study picks composites, declares what to run and measure, owns its `runs.db`, and rolls up to a verdict. See Studies.

Endpoint	Method	Purpose
`/api/study/{slug}`	GET	One study's normalized spec + effective status.
`/api/study-create`, `/api/study-create-from-composite`, `/api/study-create-from-run`	POST	Create a study, or seed one from a composite/run.
`/api/study-baseline-add`, `/api/study-baseline-remove`	POST	Manage baseline composites.
`/api/study-variant-add`, `/api/study-variant-set-params`, `/api/study-variant-delete`	POST	Manage parameter-override variants.
`/api/study-intervention-add`, `/-update`, `/-delete`	POST	Manage text-only experimental conditions.
`/api/study-readouts`, `/api/study-readout-migrate`, `/api/study-observable-check`, `/api/study-verify`	GET/POST	Declare/verify readouts against what the composite actually emits.
`/api/study-run-baseline`, `/api/study-run-variant`	POST	**Run** — build the composite in-process, merge params, record the trajectory in `runs.db`.
`/api/study-run-delete`, `/api/study-runs-clear`, `/api/study-sync-runs`	POST	Delete a run, clear runs, or reconcile `runs.db` → yaml.
`/api/study-grade`, `/api/study-tests-run`, `/api/run-tests`	POST	Compile a run into per-test verdicts. `study-grade` is the fast, no-re-sim path.

Endpoint	Method	Purpose
`/api/study-test-audit`, `/api/study-audit`, `/api/study-rigor`	GET	Tests-sufficiency audit, reproducibility audit, rigor scorecard.
`/api/study-findings`, `/api/study-findings-populate-observations`	POST	Author findings; fill quantitative finding slots deterministically.
`/api/study-set-analyses`, `/api/study-analysis-outputs`, `/api/study-analysis-file`, `/api/study-analysis-zip`	GET/POST	Post-run Analysis Steps + their outputs.
`/api/study-charts/{slug}`, `/api/study-behavior-card/{slug}`, `/api/study-refresh-viz/{name}`	GET/POST	Rendered charts and the behavior-test report card.
`/api/study-report-single`, `/api/study-reproduce`, `/api/study-rename`, `/api/study-export`	GET/POST	Single-study report, reproduce, rename, export.
`/api/study-seed-followup`, `/api/study-narrative-command`	POST	Seed a follow-up study; narrative writes.
`/api/study/{slug}/figures.zip`, `/outputs.zip`, `/notebook`	GET	Bundled downloads.

Two run engines behind the study spine

A **scratch run** (`/api/composite-test-run`) is detached and durable — it returns `202` and outlives the request. A **study run** (`/api/study-run-baseline` / `-run-variant`) runs *synchronously inside the HTTP request* today and owns the durable science in `studies/<slug>/runs.db`. Both join the dashboard's `runs_meta` table to the engine-written trajectory on `run_id`.

Investigations

A DAG of studies under one research question (**~41 routes** at HEAD). Since PR #715 an investigation compiles into a process-bigraph composite so the real scheduler orders execution.

Endpoint	Method	Purpose
`/api/investigations`	GET	List investigations.
`/api/investigation/{slug}`	GET	One investigation (+ `/report`, `/figure/{n}.{ext}`, `/figures.zip`, `/figures-build`).
`/api/investigation-create`, `/api/investigation-clone`, `/api/investigation-delete`	POST	Lifecycle.
`/api/investigation-graph`, `/api/investigation-state-tree`, `/api/investigation-summaries`	GET	The DAG canvas, the compiled state tree, roll-up summaries.
`/api/investigation-composite`, `/-add`, `/-doc`, `/-perturb`, `/-rebuild`	GET/POST	The investigation-as-composite: inspect, add members, rebuild the compiled document.
`/api/investigation-run`, `/-run-one`, `/-run-unblocked` (+ `-status`), `/-rerun`, `/-trigger` (+ `-status`)	POST/GET	Run all, one, or just the unblocked members; pull-or-compute triggers.
`/api/investigation-comparison`, `/-add`, `/-update`	GET/POST	Baseline + variant overlays for comparison charts.
`/api/investigation-add-viz`, `/api/investigation-render-viz`, `/api/investigation-viz-html`	POST/GET	Investigation-level visualizations.
`/api/investigation-report/{slug}`, `/api/investigation-rigor`	GET	Rendered report; investigation rigor.

Endpoint	Method	Purpose
`/api/iset-close`	POST	Close an investigation → render report, stamp `status: closed`, open a PR. **Never auto-merges.**

Visualizations

Declared Visualization Steps render to HTML after a run. `/viva-viz` authors them through a request/response handoff on disk.

Endpoint	Method	Purpose
`/api/visualization`, `/api/visualization-classes`, `/api/visualization-instances`	GET	The viz catalog and instances.
`/api/visualization-create`, `/api/visualization-generate`	POST	Request generation of a new viz (writes `.pbg/viz-requests/<name>.md`).
`/api/visualization-preview`, `/api/visualization-preview-instance`, `/api/visualization-status`	GET/POST	Preview a generated viz before committing.
`/api/visualization-accept`, `/api/visualization-add-to-project`, `/api/visualization-commit-batch`	POST	Stage + commit accepted viz code.
`/api/saved-visualizations`	GET	Saved interactive viz (3D packs, PTools cards) for the Analyses tab.
`/api/loom-savepoint`, `/api/loom-savepoints`	POST/GET	Save-points for the embedded `bigraph-loom` state-tree viewer (`/loom-explore`).

Analyses & run data

The Runs tab indexes runs across every emitter backend; the Analysis tab hosts saved interactive visualizations, 3D viewers, and the Analysis Tools / PTools card. (The standalone no-code Data Explorer was removed — see the note below.)

Endpoint	Method	Purpose
`/api/simulations`, `/api/simulation`, `/api/simulation-run`, `/api/simulation-run-download`	GET	The Simulations DB / runs index.
`/api/data-sources`, `/api/data-source-file`	GET	Emitter-backend data sources.
`/api/observables`, `/api/observable`, `/api/generation`, `/api/study-results`, `/api/study-bigraph-paths`	GET	Observable listing and per-run result reads that back the explorer panels.
`/api/analysis-tools`, `/api/analysis-viewers`, `/api/analysis-viewer/{uid}/launch`	GET/POST	The Analysis Tools catalog and external viewers (e.g. PTools).

> ⚠ **Accuracy note — the `/api/explorer/*` routes**
>
> The design doc `docs/data-explorer.md` and older surveys describe explorer endpoints `GET /api/explorer/{runs,observables,flux,vector}` and `POST /api/explorer/series`. **These are not registered in `api/app.py` at the verified HEAD** — the "Data explorer routes" section is now empty, and the explorer data appears to be served through `/api/observables`, `/api/observable`, `/api/generation`, and `/api/study-results` instead. Verify against `GET /openapi.json` on your server before depending on an `/api/explorer/*` path.

Remote / GovCloud

Off-load compute to **viva-api** (the GovCloud simulation backend). Point the server at it with `VIVA_API_BASE` (fallback alias `SMS_API_BASE`), then submit runs that execute remotely (Ray → AWS Batch → zarr/parquet on S3) and land back as study runs. Reaching a GovCloud endpoint needs an SSM tunnel. Backs `/viva-remote-run`.

Endpoint	Method	Purpose
`/api/remote-run-build`, `/api/remote-run-pinned-build`	POST	Build (or pin-build) a simulator on viva-api.
`/api/remote-run-submit`, `/api/remote-run-start`	POST	Submit a run.
`/api/remote-run-land`, `/api/remote-run-land-artifacts`, `/api/remote-run-analysis`	POST	Land results + artifacts + analyses locally.
`/api/remote-run-poll`, `/api/remote-run-status`, `/api/remote-run-config`	GET	Poll a submitted run.
`/api/remote-run-chain-progress`, `/api/remote-dispatch-preflight`	GET	Chain progress; pre-dispatch validation.
`/api/remote-analysis-figure(s)`, `/api/study-remote-figures`	GET	Remotely-produced figures.

Git & workstream

Every write already commits; these manage branches, PRs, and status (the GitHub Branches tab).

Endpoint	Method	Purpose
`/api/work-start`, `/api/work-end`, `/api/work-push`, `/api/work-create-pr`, `/api/work-link-branch`, `/api/work-attach-report`	POST	Workstream lifecycle → branch → PR.
`/api/work-status`, `/api/work-composite-diff`	GET	Current workstream + composite diff.
`/api/git-status`, `/api/dirty-status`, `/api/dirty-commit-all`, `/api/branch/push`	GET/POST	Working-tree status, commit-all, push.
`/api/github-repo`, `/api/auth/github/{start,poll,status,token,orgs,logout}`	GET/POST	GitHub device-flow auth.

References, catalog & suggest

Endpoint	Method	Purpose
`/api/registry`, `/api/registry/process-template`, `/api/registry/run-process`	GET/POST	The discovered Process/Step registry; run a process interactively.
`/api/catalog`, `/api/catalog-install`, `/api/catalog-uninstall`, `/api/catalog-uninstall-impact`, `/api/marketplace`	GET/POST	The module catalog + marketplace. Backs `/viva-catalog`.
`/api/module-import-diagnostics`, `/api/ecosystem-index`, `/api/framework-metrics`	GET	Import diagnostics; ecosystem federation index; metrics.
`/api/references-bib`, `/api/reference-bibtex`, `/api/reference-pdf`, `/api/dataset`, `/api/expert-doc`, `/api/expert-search`	GET/POST	Bibliography, PDFs, datasets, expert-doc search.
`/api/report-lint`, `/api/citation-gaps`, `/api/band-provenance`	GET/POST	Report linter, uncited-band gaps, band provenance (the `cite-bands` write path).
`/api/finding`, `/api/conclusion`, `/api/decision`, `/api/evidence`	POST	Narrative/decision writes.
`/api/suggest`, `/api/suggest-poll`	POST/GET	The "Suggest" button → `/viva-suggest`.

The dashboard-API vs viva-api split

Two services, two layers, joined on one key.

	Dashboard API (`api/app.py`)	**viva-api (GovCloud)**
Role	Reads the **local workspace** and orchestrates	Remote **compute + build + storage**
Owns	investigations, studies, composites, registry, charts, catalog, references	simulator builds, run lifecycle, result storage/streaming
Paths	flat `/api/<resource>` (FastAPI + pydantic v2)	versioned `/core/v1/simulator/*`, `/api/v1/simulations/{id}/*`, `/compose/v1/*`
Join key	`SimRow` (local index entry)	`SimulationRun` (cloud compute record)

The dashboard **consumes** viva-api, never duplicates it — its hand-written client is `lib/sms_api_client.py` (config `VIVA_API_BASE` / `SMS_API_BASE`, default `http://localhost:8080`). A run submitted remotely lands back as a study run in the local workspace, and the two records are joined on `run_id`.

> **Design vs shipped**
>
> The survey (`docs/workbench-api-survey.md`) is explicit that it "records what the API *does* today, not what it should" — some routes are accidental, some design specs (`WorkspaceStore`, `SessionRegistry`, a single `RunBackend`) are only partly landed. Where a route's name here differs from a live server, trust `GET /openapi.json`.

Next: On-disk schema reference

reference schema study workspace

On-disk schema reference

Everything in the Vivarium ecosystem is a file on disk. A workspace is a git repository; the objects inside it — the workspace manifest, its studies, its investigations, its composites — are YAML and JSON documents that you author and the Workbench reads, writes, and commits. This chapter is the field guide to those shapes: what each file must contain, what it may contain, and how the schemas have drifted across versions.

Three documents carry almost all the structure:

File	What it is	Authoritative schema
`workspace.yaml`	the repository manifest — one per workspace	`viva_superpowers/schemas/workspace.schema.json` (Draft-07)
`study.yaml`	one question wrapped around a composite	migrated on load; no frozen JSON schema (see note)
`investigation.yaml`	a collection of studies under one research question	migrated on load; no frozen JSON schema (see note)

For the concepts these files encode — Composite, Study, Investigation, Run, Finding, Verdict — read Core concepts first. This chapter is the reference; that chapter is the argument.

On this page

What's here the field guide to the on-disk shapes — `workspace.yaml`, `study.yaml`, and `investigation.yaml`: what each file must contain, what it may, and how the schemas have drifted across versions. · **See also** Core concepts for what these files encode, and Schemas, types & state for the type system underneath.

> **A word on `viva` vs `pbg`**
>
> The ecosystem was renamed from **pbg** to **viva**, and the migration is mid-flight. Prefer the `viva` spelling, but expect exceptions on disk: the Python package is `viva_<pkg>/` in new scaffolds (older ones use `pbg_<pkg>/`), spec ids are written `viva_<slug>.composites.<name>` (older schemas show `pbg_<slug>...`), and **the runtime control directory is still `.pbg/`** — that path did not change. See The stack.

`workspace.yaml` — the repository manifest

The manifest is what turns an ordinary git repository into a workspace. Its schema is the one file in the ecosystem that is genuinely authoritative — it is validated on every load and save by `viva_superpowers/workspace_yaml.py` (`load_workspace` / `save_workspace` / `validate_workspace`) against `workspace.schema.json`.

Required fields

Only four keys are required. A manifest with just these is valid.

Field	Type	Notes
`schema_version`	integer, `2` **or** `3`	Current accepted versions. `3` adds `default_baseline`.
`name`	string	The workspace name (non-empty).
`created`	string, `date`	ISO date, e.g. `2026-09-23`.
`plugin_version`	string, semver	The `viva-superpowers` version that scaffolded it (`^\d+\.\d+\.\d+$`).

Key optional fields

Everything else is optional and additive. These are the ones you will actually author:

Field	Type	What it does
`package_path`	string	Path to the workspace's Python package (the `viva_<pkg>/` dir) exposing `build_core()`.
`default_baseline`	object	v3: pre-fills a composite + params + run knobs so new studies don't repeat them (`composite`, `params`, `duration_s`, `interval_s`, `stop_on`, …).
`imports`	map	Imported `viva-*` / `pbg-*` repos whose composites this workspace instantiates directly (see below).
`expert_docs`	list	Expert PDFs/notes: `{name, path, sha256?, contributor?, claims_supported[]}`.
`observables`	list	Named store readouts: `{name, store_path, units?, description?}`.
`visualizations`	list	Declared figures: `{name, class?, type?, observables[], config?}`.
`simulations`	list	Named run configs: `{name, t_start, t_end, composite?, initial_state?, parameter_overrides?, emitter_config?}`.
`datasets`	list	`{name, path?, url?, sha256?, claims[]}`.
`references_bib`	string	Path to the shared `.bib`.
`references_pdfs`	list	`{bib_key, path, sha256}`.
`server`	object	`{enabled: bool}`.
`ui`	object	`{composite_view: "loom-explore" | "bigraph-viz"}` — which renderer draws composite wiring.

Field	Type	What it does
`layout`	object	Per-directory relocation overrides (`studies`, `investigations`, `composites`, `reports`, …). Omit it to keep the classic flat layout.

> ⚠ **Accuracy note — `runtime:` is documented but not schema-validated**
>
> The concept docs describe a `runtime:` block (`default_emitter: parquet|sqlite`, `subprocess_timeout_s`, `shared_artifacts: [...]`). The root object in `workspace.schema.json` does **not** forbid extra keys, so a `runtime:` block loads without error — but it is **not** a defined property in the schema_version 2/3 schema. If you rely on it, confirm your `viva-superpowers` version reads it. The `default_emitter` choice is what selects the SQLite vs Parquet emitter — see Emitters.

`imports` is worth its own note because it is how one workspace pulls in another repo's processes. Each entry is keyed by name and shaped:

```
imports:
  v2ecoli:
    source: vivarium-collective/v2ecoli   # required
    ref: main                             # required — branch, tag, or commit
    mode: reference                       # required — reference | fork-source |
in-place
    path: external/v2ecoli                # optional
    installed: true                       # true once pip-installed into the
venv
    install_path: external/v2ecoli        # what `pip install -e` was pointed at
```

An import is only discoverable by the engine once `installed: true` — `allocate_core()` discovers processes by scanning **pip-installed** distributions that declare `bigraph-schema` as a dependency. See Install & deploy.

Annotated skeleton

```yaml
# workspace.yaml — the four required keys, then the common optionals
schema_version: 3                          # 2 or 3
name: my-project
created: 2026-09-23
plugin_version: 0.22.0

package_path: viva_my_project              # the viva_<pkg>/ dir with
build_core()
default_baseline:
  composite: viva_my_project.composites.baseline
  params: {media: glucose}
  duration_s: 3600
  interval_s: 1

imports:
  spatio_flux:
    source: vivarium-collective/spatio-flux
    ref: main
    mode: reference
    installed: true

observables:
  - {name: dry_mass, store_path: cell/mass/dry, units: fg}

references_bib: references/papers.bib
server: {enabled: true}
```

Legacy keys you may still see

`pbg_processes: []` and `stages: {}` are held over from the v0.1.x nine-stage flow. They remain optional so migrated workspaces validate; new workspaces omit them.

`study.yaml` — the v4 narrative spine

A study is one question asked of one composite: run it under this configuration, measure these readouts, and turn the result into a verdict, a figure, and a report card. On disk it is a directory (`studies/<slug>/` or, in current scaffolds, `investigations/<inv>/studies/<slug>/`) whose `study.yaml` is the spec.

> ⚠ **Accuracy note — no frozen JSON schema**
>
> Unlike `workspace.yaml`, there is **no** `study.schema.json` in the codebase. Specs are **versioned (v2 → v3 → v4) and migrated on load** by `vivarium_workbench.lib.spec_migration` and `investigations.load_spec()`; the execution-relevant subset is normalized by `study_spec.study_interface()` into `{composite, config, inputs, outputs, emitter}`. The section names below are faithful to the framework's own reference material, but exact field spellings drift between versions. **Confirm field names against a live workspace** (or the loader) before depending on them, and remember that only *declared* fields render — every field is optional.

The v4 spec organizes into six groups. Read top to bottom, it is the life of one study.

Framing — what you are asking

```
purpose:
  question: Does intrinsic DnaA-ATP hydrolysis alone keep the ATP fraction in
band?
  mechanism: Splits the DnaA pool into apo / DnaA-ATP / DnaA-ADP; wires
intrinsic hydrolysis.
  expected_outcome: failing-bio — intrinsic hydrolysis alone cannot reach [0.2,
0.5].
hypothesis: The reset network (RIDA/datA/DARS) is required, not optional.
key_assumptions:
  - Single-seed trajectory is representative for this qualitative claim.
conditions:                              # grouped replacement for baseline +
variants
  baseline: {composite: viva_ecoli.composites.baseline}
  model_settings:                        # tunable-parameter catalog
(value/default/range/cite)
    - {name: k_hydrolysis, value: 0.046, units: 1/min, cites: [Boesen2024]}
```

Simulations — what to run

```
simulation_set:                          # v4 replaces v3 `variants:` +
`interventions:`
  - {name: baseline, base_model: baseline, condition: rich, seeds: [0, 1, 2],
duration_s: 3600}
```

Build — what to implement

```
model_change: Add intrinsic hydrolysis MONOMER0-160 → MONOMER0-4565.
implementation_requirements:
  - A DnaA nucleotide-state process with apo/ATP/ADP stores.
build_plan: [Port the hydrolysis rate, wire the three stores, smoke-run 100
steps.]
inputs_to_supply: []
```

Validation — how you will know

```
readouts:                                  # v4 replaces v3 `observables:`
  - {name: datp_fraction, store_path: cell/dnaa/atp_fraction, status: emittable}
behavior_tests:
  - name: datp_in_band
    classification: primary                # primary | supporting | diagnostic |
regression
    measure: {kind: generation_average, path: cell/dnaa/atp_fraction}
    pass_if: {op: in_range_every_generation, low: 0.2, high: 0.5}
    cites: [Boesen2024]
biological_summary: >
  DnaA cycles between ATP- and ADP-bound states; initiation competence tracks
the
  ATP-bound fraction.
```

Report & Decide — the conclusion

```yaml
report:
  verdict: failing-bio                    # passing | passing-with-caveats |
failing-bio |
  confidence: Investigating               #   failing-impl | inconclusive | not-
yet-run
  evidence_quality: single-seed
  main_insight: Intrinsic hydrolysis alone undershoots the band.
  caveat: Qualitative; not yet replicated across seeds.
findings:
  - id: hydrolysis-insufficient
    statement: The ATP fraction settles below 0.2 without the reset network.
    kind: biological                      # biological | computational |
methodological
    status: confirms                      # confirms | partial | contradicts |
novel
    provenance: {run_ids: [run-0007], model_commit_hash: abc123}
conclusion_logic:
  if_pass: Proceed to the box-binding study.
  if_fail: Seed a reset-network study (this is the expected branch).
pipeline_gate:
  prerequisites: [dnaa-01-expression-dynamics]
  enables: [dnaa-03-box-binding]
  proceed_condition: tests-passed
```

Provenance — the audit trail

```yaml
seeded_from: dnaa-01-expression-dynamics
references: [{file: Boesen2024.pdf, section: "Fig 3"}]
runs:                                     # back-compat mirror of runs.db
  - {run_id: run-0007, outcomes: {datp_in_band: {result: FAIL}}}
visualizations:
  - {name: DnaANucleotideTrajectory, address: local:DnaANucleotideTrajectory}
```

Reserved v4 field names

Schema v4 reserves `tests` (object), `references` (list of `{file, section}`), and `implementation_tasks` (string). A v3 spec with a differently-shaped field of the same name collides during v3 → v4 migration. The standard renames are `references:` (dict) → `bibliography:` and `implementation_tasks:` (list) → `tasks:`. Setting `schema_version: 4` short-circuits the migration.

> **Derive-on-read**
>
> A study renders **Ran · Tests N ✓ · Passed** only when `runs[].outcomes` (backed by `runs.db`) support the test results. A test with an authored `status: passed` but no run outcome renders as pending — this is what stops a study from *claiming* a result it never produced. See Studies and Rigor & evidence.

`investigation.yaml` — the 9-section spine

An investigation is a named collection of studies building one cumulative argument under a shared question. On disk it is `investigations/<slug>/investigation.yaml` — and the slug is also a git branch and a worktree, so parallel agents never trample each other.

> **Accuracy note**
>
> As with `study.yaml`, there is no frozen JSON schema; investigations are `schema_version: 1` (minimal) or `2` (the current narrative spine), migrated on load. The member-list key is read by `investigation_member_slugs()`, which accepts `studies:` (pre-migration) or `members:` (post-migration), preferring `studies:` when both are present. Confirm field names against a live workspace.

The spine groups into five parts:

Group	Fields	Purpose
Identity	`schema_version`, `name`, `title`, `created`, `status`	What this is.
Framing	`question`, `hypothesis`, `lead`, `biological_story`, `glossary`	Why it exists.
Membership	`studies[]` / `members[]`, `at_a_glance`, `acceptance_criteria[{study, behavior}]`, `parts[]`	What it contains.
Synthesis	`executive`, `scientific_argument`	What it concludes.
Review	`feedback_rounds[]`, `expert_docs[]`	How it was reviewed.

```yaml
# investigation.yaml (schema_version 2)
schema_version: 2
name: dnaa-replication
title: DnaA-driven replication initiation in whole-cell E. coli
created: 2026-09-23
status: in-progress

question: Which DnaA regulatory mechanisms are required to hold the ATP fraction
in band?
hypothesis: Intrinsic hydrolysis is insufficient; the extrinsic reset network is
required.
lead: eagmon

executive:                                # state-first opening, single-sourced
into the report
  what_is_this: A staged port of the DnaA nucleotide cycle into the baseline
cell.
  verdict: The reset network is required.
  verdict_status: in-progress             # in-progress | passed | complete |
blocked | failed | planning
  decisions_needed: []

members: [dnaa-00-parameter-foundation, dnaa-02-atp-hydrolysis, dnaa-03-box-
binding]
acceptance_criteria:
  - {study: dnaa-02-atp-hydrolysis, behavior: datp_in_band}

parts:                                     # optional — group members into
narrative phases
  - name: "II. Nucleotide cycle"
    overview: Split the pool into apo / ATP / ADP and wire the cycle.
    studies: [dnaa-02-atp-hydrolysis]

expert_docs:
  - {name: Boesen 2024, path: references/Boesen2024.pdf}
```

The dependency DAG is computed, not stored

An investigation stores **no edges**. The cross-study DAG is computed at render time from each member's `pipeline_gate.prerequisites`. Since August 2026 an investigation is also *compiled* into a process-bigraph composite — each study becomes a `StudyStep` node and each prerequisite edge becomes store wiring, so the real scheduler orders execution. A blocked prerequisite prunes the dependent member rather than a scheduler deciding "skip this." See Investigations.

The on-disk workspace layout

Putting it together, a workspace directory looks like this:

```
my-project/                                  # a git repository
├── workspace.yaml                           # the manifest (schema above)
├── viva_my_project/                         # the Python package
(package_path)
│   ├── core.py                              #   build_core() — the
type/process registry
│   ├── composites/<id>.composite.json       #   runnable substrate (JSON or a
generator .py)
│   ├── processes/                           #   Process / Step classes
│   └── visualizations/                      #   @as_visualization /
Visualization steps
├── investigations/
│   └── <inv-slug>/
│       ├── investigation.yaml               #   the collection (= git branch
= worktree)
│       ├── studies/<study-slug>/study.yaml  #   nested studies (current
convention)
│       └── reports/                         #   per-investigation report HTML
├── studies/                                 # flat studies (legacy path —
still resolves)
│   └── <study-slug>/
│       ├── study.yaml                       #   the question + spine
│       ├── runs.db                          #   canonical run + outcome
record (SQLite)
│       └── parquet-runs/<run>/              #   emitted trajectories (Parquet
hive)
├── references/papers.bib                    # shared bibliography
├── notes/                                   # field records — cleanup PRs
must SPARE these
└── .pbg/                                     # runtime control dir (NOT
renamed to .viva/)
    ├── server/                              #   dashboard runtime state
(server-info, pid, log)
    ├── artifacts/<hash>/                    #   content-addressed artifact
store
    └── worktrees/<inv-slug>/                #   per-investigation worktrees
```

Two runtime artifacts are worth calling out because you will read them directly:

- **`runs.db`** — a per-study SQLite database. Its `runs_meta` table is the authoritative provenance record (`run_id`, `started_at`, `completed_at`, `composite`, `variant`, `emitter_path`, `generation_id`). Every run appends here; the derive-on-read

verdict reads from it. Some visualizations (3D viewers, nested coordinate arrays) open it directly.

- `parquet-runs/<run>/` — the emitted trajectories, when the workspace's emitter is Parquet. The SQLite emitter instead writes full per-step state into `runs.db`.

> **The relocatable layout**
>
> A workspace may move any of these directories via the `layout:` block in `workspace.yaml`; tooling resolves paths through `WorkspacePaths`, never by hardcoding the names. The tree above is the conventional flat default.

Schema-version drift at a glance

Because the ecosystem is mid-migration, the three documents version independently:

Document	Versions	What later versions add
`workspace.yaml`	**2 → 3**	v3 adds `default_baseline`.
`study.yaml`	**2 → 3 → 4**	v3 adds the canonical body; v4 adds the narrative spine and reserves `tests` / `references` / `implementation_tasks`.
`investigation.yaml`	**1 → 2**	v2 adds the 9-section narrative spine and the `executive` block.

Legacy v2 studies live at `investigations/<name>/spec.yaml` and are migrated in memory on read. When in doubt, the **loader is the authority**, not a frozen list.

Next: Install & deploy

getting-started reference workspace

Install & deploy

Getting a workspace and the Workbench running — on your laptop, in a container, and as a read-only site others can browse. The through-line is one fact worth fixing in your mind before anything else:

> **The direction of the dependency**
>
> **The workspace depends on the Workbench, not the other way around.** `vivarium-workbench` is a plain pip dependency of your workspace's `pyproject.toml`, installed into the *workspace's own virtualenv* and run from there. The dashboard imports your workspace's package (`build_core()`) and any installed simulation stacks to build and run composites — a dashboard installed in some other environment could not see them.

That is why there is no global "install the Workbench" step. You scaffold a workspace, and the Workbench comes along inside its venv. For the layers involved, see The stack.

> **On this page**
>
> **What's here** what to install and how to run it — prerequisites, the local laptop path, the container image, and a read-only site others can browse. · **See also** The stack for the layers involved, and the worked example to see the setup used end to end.

Prerequisites

- **uv** — the package manager the ecosystem standardizes on. It creates the workspace venv and resolves the git-sourced dependencies.
- **git** — a workspace *is* a git repository, and every Workbench action commits to it.

- **Python 3.12** — the version the server image is built against.
- **Node + npm** — only needed if you build the container image yourself (the vendored `bigraph-loom` bundle is built with npm); not needed for local use.

The two things every dashboard-touching skill assumes are a **workspace** (a directory with `workspace.yaml` + a `viva_<pkg>/` package) and a **running server**. The rest of this chapter is how to get both.

Local start

From inside a workspace directory:

```
# 1. create the workspace venv and install everything (resolves the git pins)
uv venv .venv && source .venv/bin/activate
uv pip install -e ".[dev]"

# 2. sanity check the workspace
python3 scripts/lint-workspace.py        # → "workspace lint: OK"

# 3. serve the dashboard against this workspace
vivarium-workbench serve --workspace . --host 0.0.0.0 --port 8000
```

`vivarium-workbench` renders the workspace once, then serves the SPA + HTTP API until Ctrl-C. It writes its base URL to `.pbg/server/server-info` — skills read the URL from there rather than hardcoding it.

> **`vwb` is the short alias**
>
> The CLI is aliased `vwb`, so `vwb serve --workspace .` is equivalent. Omitting `--port` picks a free port and prints the URL; a scaffolded workspace also ships a `scripts/serve.sh` wrapper that prefers the workspace venv's binary. The `/viva-workbench` skill wraps all of this (`/viva-workbench start`).

> **Renamed from `vivarium-dashboard`**
>
> The distribution was `vivarium-dashboard`; it is now `vivarium-workbench`. The old `vivarium-dashboard` / `vdash` / `vivarium-dashboard-publish` CLIs and the `VIVARIUM_DASHBOARD_*` env vars keep working as deprecated aliases during the migration window. The one consumer-facing change is the **dependency name** in `pyproject.toml`.

Working on the Workbench itself

If you are hacking on the server, do not edit the committed git pin. Override it with an editable install into the workspace venv, pointing at your local clone:

```
# from inside the workspace, with its venv active
uv pip install -e /path/to/vivarium-workbench
vivarium-workbench serve --workspace .    # now runs your working copy
```

Scaffolding a workspace

New workspaces are scaffolded from the **viva-template** repo. The template's `template-init.sh` renders its `.j2` files (the `pyproject.toml`, the `viva_<pkg>/core.py` with `build_core()`, the `.pbg/schemas/` validators) into a concrete workspace. The `/viva-workspace` skill drives this in three modes:

```
# standalone — clone viva-template directly
/viva-workspace my-project

# upstream-branch — clone an existing repo and lay a workspace branch on top
/viva-workspace my-project --upstream owner/repo

# in-place — promote an existing git checkout (e.g. a composite-only repo) into
a workspace
/viva-workspace my-project --target . --in-place
```

Under the hood the standalone mode runs `vwb scaffold-workspace --name --target` (which clones the template), `uv venv .venv`, `uv pip install -e .[dev]`, commits the bootstrap, and registers the workspace in the global catalog at `~/.pbg/workspaces.json`.

> **Template repo naming**
>
> The scaffold repo is `viva-template` (formerly `pbg-template`); some older docs and the `$PBG_TEMPLATE` / `--template-source` override still reference the `pbg-template` name. Both point at the same scaffold.

Because the Workbench is **not on PyPI during beta**, the scaffolded `pyproject.toml` pins it to a git source, so CI, Docker, and collaborators all resolve identically:

```
[project]
dependencies = ["process-bigraph", "bigraph-schema", "vivarium-workbench",
"..."]

[tool.uv.sources]
vivarium-workbench = { git = "https://github.com/vivarium-collective/vivarium-
workbench.git", branch = "main" }

[tool.hatch.metadata]
allow-direct-references = true
```

The git ref can be overridden at init via `VIVARIUM_DASHBOARD_REF`.

The module catalog

Simulation stacks (`viva-*` / `pbg-*` wrapper repos) are installed into the workspace venv as **community modules**. The curated catalog lives in `viva_superpowers/catalog/modules.json`; the `/viva-catalog` skill lists, installs, and uninstalls entries through the Workbench API:

```
/viva-catalog                 # list installed + available modules (default)
/viva-catalog install <pkg>   # add a module to the workspace
/viva-catalog uninstall <pkg> # remove one
```

Modules are installed as git submodules / editable git installs, and a module only becomes discoverable once it is pip-installed into the venv and declares `bigraph-schema` as a dependency — that is how `allocate_core()` finds and registers its processes without any manual `register_link()` call.

> ⚠ **The `--no-deps` install traps**
>
> When baking modules into a container image (or installing several at once), install them with `pip install --no-deps`. Two failure modes make this non-optional:
>
> 1. **The giant image.** Resolving a heavy stack's full dependency tree can drag in a multi-gigabyte GPU/ML chain (nvidia, torch, triton, ray) that the Workbench never touches — the server renders composites and spawns workers; it does not run the sims.
> 2. **The dependency-name mismatch.** Mid-rebrand, a module's own metadata may still require the old distribution name (e.g. `pbg-ketchup`) while the installed package is the new one (`viva-ketchup`). A full re-resolve then either fails or pulls a second, skewed copy. `--no-deps` sidesteps both by installing exactly the wheel you name and nothing else.
>
> The trade-off is that you must then add any genuinely-needed companion (like the `pbg-ptools` viewer plugin) by hand — also `--no-deps`.

Docker

The Workbench ships a container image that is **the tool, not the science environment**. It contains the server and its own declared dependencies (built from the repo's own `uv.lock`), and deliberately does *not* bake in a workspace package or a compute stack. At runtime the workspace — including its own `.venv` — is mounted at `/workspace`, and the server spawns env workers into that interpreter.

```
# build + push (linux/amd64; the deploy nodes are x86_64)
deploy/build-and-push.sh [version] [org]
#   → ghcr.io/vivarium-collective/vivarium-workbench:<git-sha>

# run locally against a mounted workspace
docker run --rm -p 8000:8000 \
  -v /path/to/workspace:/workspace \
  ghcr.io/vivarium-collective/vivarium-workbench:<tag>
#   the image's default CMD is: serve --workspace /workspace --host 0.0.0.0 --
port 8000
```

> **The workspace must bring its own venv**
>
> The image sets `VIVARIUM_WORKBENCH_REQUIRE_WORKSPACE_VENV=1`: the environment resolver refuses to silently fall back to the thin server venv for workspace work (which could not import the workspace package). The mounted `/workspace/.venv` is required, and the seam fails loudly if it is missing.

Cutting a semver release

```
deploy/bump-and-release.sh <version> [org]   # e.g. 0.3.86
```

This bumps `pyproject.toml`, commits, creates an annotated `v<version>` tag, pushes, and dispatches the build against `main` — **refusing** unless the tree is clean, `main` is checked out and in sync with `origin/main`, `<version>` is a real semver increase, and neither the tag nor the image already exists. A bare short-sha build (the default) is unrestricted. Every image records the commit it was built from in OCI labels and `/app/BUILD_INFO.json`.

Kubernetes / cloud deployment

The container image is owned by the `vivarium-workbench` repo; the **Kubernetes manifests live in the viva-api repo** (the renamed `sms-api`), where the Workbench is deployed as a peer service of viva-api — one deploy brings both up. Salient facts for anyone standing it up:

- **Single replica by design.** The Workbench serves one workspace directory that it reads *and writes* (git commits, `runs.db`), so it runs with its own **RWO** `gp3` PVC and a single replica / `Recreate` strategy — no horizontal scaling.
- **The workspace is a mounted volume**, seeded once from the baked-in workspace by an initContainer, then owned by the running pod.
- **No new secrets.** The Workbench delegates all heavy compute/storage to viva-api over in-cluster HTTP (`SMS_API_BASE` → the in-cluster service DNS); it needs no Postgres, no IRSA, no AWS creds.

Auth and the reverse proxy

The Workbench has **no login wall and no role-based access control**. Its only auth is a GitHub OAuth **device flow** used purely for git operations on the Branches tab, plus a CSRF/origin guard on mutating routes (requests with no `Origin` — curl, the local CLI, the skills — are allowed; a present `Origin` must match `Host`).

> ⚠ **Put access control in front of it**
>
> Because there is no login wall, anything network-reachable can drive the API. For any shared or public deployment, place a **reverse proxy that enforces authentication** in front of the server. Do not expose the raw `serve` port to an untrusted network.

The read-only remote profile

The published, browsable dashboard is **the same codebase and the same API contract** as the full Workbench — it is not a separate static exporter. It differs only by a **mutation whitelist**: writes are gated, and only a small set of run-launchers and telemetry/auth endpoints remain enabled. This is the clean way to serve an investigation's report to external reviewers over the network without handing them authoring rights.

```
# export a workspace as a self-contained static bundle (the classic read-only dashboard)
vivarium-workbench-publish --workspace /path/to/workspace --out /tmp/bundle
```

The same frontend JS runs against the bundle's `api/*.json` files instead of a live server, so a published snapshot and the live dashboard look identical. For what those reports contain, see Analyses, visualizations & report cards.

Next: A worked end-to-end example

composite getting-started study

A worked end-to-end example

This chapter follows one question all the way through the framework: from an empty directory to a graded study with a computed verdict, and on to the next study it seeds. It is a map, not a copy-paste script — each step links to the chapter that covers it in depth, and the exact command surface evolves, so confirm specifics against your installed `/viva-*` skills and a live workspace.

On this page

Assumes you've skimmed Studies and have `uv` plus the `viva-superpowers` plugin installed. · **You'll learn** how one question travels the whole loop — from an empty directory to a scaffolded workspace, a composite, graded runs, a computed verdict, and the next study it seeds.

The shape of the loop

Every step below is one hop of the discovery loop from What is Vivarium?: **Question → Composite → Run → Verdict → Next.** If you internalize that shape, the individual commands are just details.

The running example is a small microbial-growth question, in the spirit of the **Spatio-Flux** reference models and the **v2ecoli baseline** showcase: *does our cell composite hold a steady exponential growth rate under a glucose-replete medium?*

0. Prerequisites

You need `uv` and (for the AI-driven path) the `viva-superpowers` Claude Code plugin installed. See Install & deploy for the full setup, including the Docker and Kubernetes paths.

```
uv --version          # the workspace toolchain
# the /viva-* skills are provided by the viva-superpowers plugin
```

1. Scaffold a workspace and start the Workbench

A **workspace** is a directory with a `workspace.yaml` and a `viva_<pkg>/` Python package exposing `build_core()`. Scaffold one from the template, then start the server that every later step drives.

```
# scaffold (see /viva-workspace for the three modes)
#   → creates workspace.yaml + viva_<pkg>/ + investigations/ + studies/
# then start the dashboard server (alias: vwb)
vivarium-workbench serve --workspace .
```

The server writes its live URL to `.pbg/server/server-info`; agents resolve it from there rather than hardcoding a port. Open the printed URL and you land in the Workbench — one workspace per browser tab, with the side rail (Resources · Catalog · Processes · Studies · Runs · Analysis).

Two preconditions

Almost every `/viva-*` skill assumes **a workspace** and **a running Workbench**. If either is missing, fix that first — see Working with AI agents.

2. Get the processes you need

A model is assembled from Processes and Steps. You get them three ways:

Install from the catalog

Browse and install existing community modules (processes + composites + their deps) with `/viva-catalog`. This is the fastest path when the science already exists.

Wrap a simulator

Use `/viva-expert` to wrap an upstream simulator as a real, bridged `Process` — its heavy mode even spins up a sibling `viva-<name>/` repo with tests and a showcase. See the Skill reference.

Write one by hand

Subclass `Process`, implement `inputs()` / `outputs()` / `update()`, and register it with `core.register_link(...)`. See Processes & Steps.

If you only have an interface in mind but not the mechanism yet, start with a draft process: it declares the ports but stays inert and shows as **DRAFT** until you supply the dynamics.

3. Build and smoke-test a composite

Wire your processes and stores into a composite — a `state` tree of `process` / `step` nodes whose ports are wired to shared-store paths. Add an emitter so the run leaves a trajectory behind.

Smoke-test it for a few steps before attaching any science to it:

```
# run a composite from the catalog for N steps and report emitted observables
# (this is what /viva-run does under the hood: POST /api/composite-test-run)
```

A green smoke test means the wiring type-checks and the composite is ground (no unfilled required sites) — it is ready to carry a question.

4. Author the study

Now wrap the question around the composite. A study is `study.yaml`: one question, one emit-contract, one pass/fail bar. Use `/viva-study` (which canonicalizes the file and records provenance) rather than hand-editing.

```
# investigations/<inv>/studies/glucose-growth/study.yaml  (illustrative shape — confirm
# field names against a live workspace; see the Schema reference)
schema_version: 4
purpose:
  question: Does the cell composite hold steady exponential growth in glucose-replete medium?
  expected_outcome: growth rate stays within the measured band for every generation
simulation_set:
  - base_model: cell_baseline        # which composite to run
    condition: glucose_replete
    seeds: [0, 1, 2]
    duration: 3 generations
readouts:
  - name: growth_rate
    store_path: [cell, growth_rate]  # the emit contract: a name tied to an exact store path
behavior_tests:
  - name: steady_exponential_growth
    classification: primary
    measure: { kind: generation_average, path: growth_rate }
    pass_if: { op: in_range_every_generation, low: 0.30, high: 0.55 }
    cites: [SomeReference2024]
pipeline_gate:
  prerequisites: []                  # this study's place in the investigation DAG
```

The five join fields — `simulation_set`, `readouts` (the emit contract), `behavior_tests`, plus baselines and variants — are what tie the study to the composite. See Studies

for each field, and Rigor & the evidence engine for how to make a behavior test honest (an acceptance band cited to the literature via `/viva-cite-bands`).

5. Run, emit, analyze, visualize, grade

With the study authored, run it. The run advances the composite through its tick lifecycle, the emitter records the wired state, and the two-phase flush network fires: an extractor normalizes the emitted data, then analysis, visualization, and report-card Steps write their artifacts.

- **Analyses & visualizations** attach figures and tables to the run; author a bespoke one with `/viva-viz`. They surface on the Analysis rail and in each run's Results view.
- **Report cards** grade the `behavior_tests` and render a category scorecard, reused by the study's Tests tab.

The verdict is **computed from the run's outcomes, never hand-set**. A test resolves to one of:

Verdict	Meaning
✓ within_tol	measured value is inside the acceptance band
≈ drift	close, but outside tolerance — needs attention
✗ mismatch	measured value contradicts the band
– ungraded	no run yet, or the readout didn't resolve

6. Read the verdict and seed the next study

The study rolls its test outcomes up into a study **verdict** and a **readiness** signal (the count of open gaps before the verdict is trustworthy). Read it on the study page — and read *why*: divergences between what you authored and what the code

computed are the dashboard's headline (see the authored-vs-computed parallel slots in Rigor & the evidence engine).

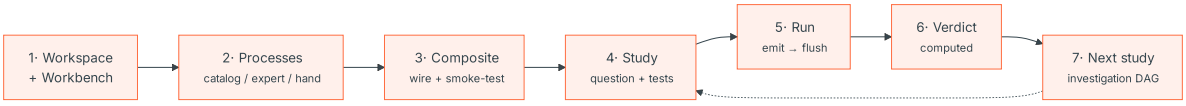
A verdict is not the end; it tells you what to ask next. If growth held, the next study might perturb the medium; if it drifted, the next study calibrates a parameter. Either way:

> New science is a **new study**, not a patch to the model.

7. Group studies into an investigation

Several such studies accumulate into an investigation — a gated DAG under one research question, living on its own git branch/worktree. Each study declares its `pipeline_gate.prerequisites`; the dependency edges are computed at render time, and since the investigation compiles to a composite, a study's prerequisites are just store wiring. When you are ready, `/viva-report` audits and renders the investigation, and the Working with AI agents division of labor applies: agents build and run in parallel; you own the framing, the verdicts, and the merge.

The whole loop, at a glance



Reference runs to study

Two published investigations are worth reading as fully-worked exemplars of this loop:

- **Spatio-Flux** — the paper's reference application: microbial-ecosystem simulations combining kinetic equations, dynamic FBA, and spatial processes. It is a standalone library built to demonstrate the composition of metabolic, spatial, mechanical, and structural processes within a single process-bigraph type system — a testbed for wiring independently developed mechanisms

together through typed interfaces. The clearest example of *composition across formalisms*.

The Spatio-Flux reference model: a composite of metabolic, spatial, mechanical, and structural processes, plus simulation snapshots of particles moving through nutrient fields.

The Spatio-Flux reference model. *(a) Metabolic, spatial, mechanical, and structural processes connected through shared typed stores: Newtonian particles carry internal metabolic processes and exchange metabolites with a spatial lattice via particle–field adapters, while diffusion updates field concentrations and graph-rewrite processes govern particle division. (b) Simulation snapshots — particles moving through and reshaping glucose, acetate, and biomass fields.* *(Agmon & Spangler, Fig 8.)*

- **The v2ecoli baseline showcase** — from raw sources to a calibrated whole cell, showing the study/investigation spine at scale.

Next: Glossary — every term in one place.

concepts glossary reference

Glossary

Every load-bearing term in one place, with its formal anchor and its legacy synonyms. The Process Bigraph vocabulary is drawn from Table 1 of Agmon & Spangler, *Process bigraphs and the architecture of compositional systems biology*; the knowledge vocabulary from the Investigation Spine framework. Where a term has an older name, it is marked **(legacy: …)** — the ecosystem is mid-migration and older documents use different words for the same idea.

Entries are grouped by the layer they belong to. Each links to the chapter that develops it.

On this page

What's here every load-bearing term in one place — its formal anchor, its legacy synonym where one exists, and a link to the chapter that develops it, grouped by the layer it belongs to. · **See also** Core concepts for the argument behind the vocabulary, and Schemas, types & state.

Process Bigraph core vocabulary

The formal spine of *what runs*. These are the entries of the paper's core-vocabulary table, in dependency order.

Path *(P)* — a name for a location in the state tree, e.g. `['cell', 'cytoplasm', 'glucose']`. Paths address where a value lives. → Core concepts

Type *(T)* — a declaration of what may be stored at a path: it knows how to produce a default, serialize, check validity, and — crucially — **apply** a delta. Types compose (`map[string, float]`, `array[(3|4), float]`, `tree[float]`), carry units, and can inherit. → Schemas, types & state

Value *(V)* — the concrete datum stored at a path: a molecule count, a concentration field, the clock reading. → Core concepts

Schema *(Σ : P ⇀ T)* — the map from paths to types; "the typed organizational blueprint of the system." The separation of schema from state is the central principle — it lets you validate, reuse, and reason about composition independently of execution. → Schemas, types & state

State *(x : P ⇀ V)* — the map from paths to values: the actual data the model holds at a moment in time. → Core concepts

Store — a state location that functions as a shared variable. Stores nest hierarchically, and that nesting *is* the place graph. Processes never talk to each other directly; they read and write shared stores, and that wiring is the coupling. → Core concepts

Process *(f : (in^τ, …) → (out^τ, …))* — a temporal edge: a mechanism with a typed interface, defined by what it reads, what it can update, its configuration, and an **interval**. Its `update(state, interval)` advances dynamics over a span of time (an ODE solve, an FBA step, a stochastic tick). → Processes & Steps

Step — a reactive edge that declares **no interval**. Its `update(state)` fires when its inputs are ready — a dataflow rule run to convergence. Emitters, analyses, visualizations, and report cards are all Steps. → Processes & Steps

Delta *(δ = update(x_in, Δt))* — the typed change a process returns. Processes emit deltas rather than overwriting state. → Core concepts

Typed application *(apply) (x′ = apply_τ(x, δ))* — the one law that merges a delta into shared state through the type's own combination rule: numeric types **accumulate**, others **set** or **merge**. Because *how* deltas combine is a property of the data type, not the process, two independently-written processes can write to the same store without knowing about each other. This is what lets them compose. → Core concepts

Port — a single input or output of an edge. A process declares typed input and output ports; wiring connects them to stores. → Composites & wiring

Wiring *(W : (process, port) → P)* — the map connecting a process's port to a store path. Wiring makes every coupling explicit and checkable — it is the link graph. →

Composites & wiring

Place graph — the containment structure: dict nesting, "cell contains cytoplasm." Distinct from the link graph (wiring). → Core concepts

Link graph — the wiring structure: ports connected to store paths. In a *process bigraph* this is a **process graph** — processes connect to stores via typed ports. → Core concepts

Type registry *(R_T)* — the collection of types the engine knows. **Link registry** *(R_L)* — the map from a process's address to its concrete handler class. Both live on the `Core`, populated by discovery (no manual registration). → Schemas, types & state

Core — the operational registry of types and process classes the engine consults, built via `allocate_core()`. → Schemas, types & state

Process bigraph *(B = (Σ, x, R_T, R_L))* — the typed, executable organization as a whole: a schema, a state, and the two registries. → Core concepts

Composite — a state-tree of typed process and step nodes wired to shared stores; **the only object the engine actually runs**. A composite is itself a Process (via its bridge), so composites nest — big models are assembled from small ones by *containment, not hole-filling*. → Composites & wiring

Bridge *(bridge : (external port) → P)* — the map that packages a composite's internal process bigraph as a single higher-level process, wiring its external interface onto internal store paths. → Composites & wiring

Orchestration *((B, t, t_next) → (B′, t′, t′_next))* — the scheduling semantics: when processes run, how their deltas are combined, and how structural rewrites are incorporated over time. The three patterns are multi-timestepping, workflow (Step DAG), and event-driven graph rewrite. → Processes & Steps

Emitter — a Step that records wired state each tick into a durable sink (SQLite, Parquet, XArray). The emitter is a study's durable phase boundary. → Emitters

Document vocabulary: site, fill, ground

The unifying idea — a composite, a template, a study, and an investigation are all the same kind of typed document, differing in structure, not in execution machinery.

Document — a `bigraph-schema`-typed object: a place graph (dict nesting), a link graph (wiring), and holes (sites). One object, one operation, one law. → The stack

Site *(legacy: slot, hole)* — a place-graph hole: a slot where a whole composite, process, or value plugs in. This is Milner's bigraph *site*. A composite refuses to run while any required site is open. → Templates & draft processes

Fill *(legacy: bind, reify, substitute)* — the one operation on documents: substitute a value or a sub-composite into a site. Gating a study and binding a template are the same mechanism. → Templates & draft processes

Ground *(`is_ground`)* — the one law: a document runs iff it has no unfilled required sites. A **ground** document is runnable; one with open sites is a **template**. → Core concepts

Template — a composite that is not yet ground — it has open sites. A **study template** fixes the analysis network and leaves the model as a site; an **investigation template** has one site per member study. → Templates & draft processes

Draft process — a template idea applied to one process: a `DraftProcess` declares a contract (its ports plus a description of the transformation it is *meant* to perform) but carries **no dynamics**. Stepped, it stays inert and never fabricates behavior; it shows in the dashboard marked `DRAFT`. Distinct from a site: a draft process is an inert *node*, a site is an empty *hole*. → Templates & draft processes

> **Legacy vocabulary**
>
> Design documents from mid-2026 use **slot / bind / reify** for what are now **site / fill / ground**. The unified-architecture plan renamed and narrowed these terms and explicitly retired *slot, bind, reify, gate evaluator, barrier, phase, flush engine*. Prefer site / fill / ground; treat the older words as synonyms when you meet them.

Knowledge vocabulary

The agentic spine — *what reasons*. These are metadata objects that reference a composite by a dotted id and record what came out. None of them is the thing that runs.

Investigation — a named collection of studies under one research question; the unit of a git branch and worktree. It stores no edges — the cross-study DAG is computed from its members' prerequisites and compiled into a composite. → Investigations

Study — the reason you run a composite: one question, one emit-contract, and one pass/fail bar wrapped around it. A study is never compiled into a composite itself — it stays metadata that points at one and reads back what came out. → Studies

Run — one execution of a composite; produces a run store of trajectories recorded to `runs.db` (and `parquet-runs/`). Runs are asynchronous — a failed run can still return HTTP 200, so check the *result*, not the status code. → Studies

Analysis — a Step (an `AnalysisStep`) that *derives* an artifact — a figure, table, CSV, or markdown — from a run's normalized results, **without** issuing a verdict. Renders under Evidence › Analyses. → Analyses, visualizations & report cards

Visualization — a curated figure (a `Visualization` Step) authored to make a finding clear. The bar is deliberately high: a bare line of one observable versus time rarely clears it. Renders under Evidence › Visualizations. → Analyses, visualizations & report cards

Behavior test — a machine-checkable spec, not prose: it names how to *measure* a result from a run (`measure` + `window`) and what *passing* means (`pass_if`), so one definition drives both execution and the dashboard's pass/fail rendering. → Rigor & evidence

Report card — a Step (a `TestStep`) that reads a run and produces pass/fail outcomes keyed by test — the raw evidence. A report card **grades** (→ Assurance › Tests); an analysis merely **derives**. → Analyses, visualizations & report cards

Acceptance band — the tolerance or direction a test accepts (e.g. "within 2%", "at most 0.02"), ideally cited to literature via band provenance. The related `gate_class`

field distinguishes a pre-stated **acceptance criterion** from a post-hoc **regression pin**, discouraging tune-to-pass. → Rigor & evidence

Acceptance criterion — an investigation-level `{study, behavior}` pair linking a criterion the investigation must meet to a member study's behavior test. → Investigations

Finding — a summary of what was seen, floored by a lifecycle (`confirms / partial / contradicts / novel`) so an unproven claim can't look settled. Carries `evidence`, `expected`, and `provenance` slots; a finding is reproducible once a fresh checkout can regenerate it. → Rigor & evidence

Verdict — a study's conclusion state, **computed from its evidence, not asserted**. The evaluator rolls run outcomes and prerequisites into `passed / failed / needs_calibration / blocked / not_started` (passed iff fail == 0 and pass > 0), writing a coded slot parallel to the authored one. → Rigor & evidence

Provenance — the audit trail: every Workbench write is a git commit, so verdict, readiness, and debts are diff-able. A test's `requires_simulation` + `cites` bind a verdict to a specific run and to the literature. → Rigor & evidence

Epistemic debt — an open question the investigation owes: an item bucketed as unresolved rather than fabricated into a verdict. Part of a study's knowledge object. → Rigor & evidence

Readiness — a report-linter score that counts open gaps ("6 gaps" style), telling you exactly what is missing before a verdict is trustworthy. Because a study is typed data, hardening it is a transformation an agent can apply and verify. → Rigor & evidence

---

Naming: pbg vs viva

Prefer `viva`; expect `pbg` on disk

The ecosystem was renamed from **pbg** ("process-bigraph") to **viva**. The migration is real but not complete, so both spellings appear:

- **Skills** are `/viva-*` (older `/pbg-*` names still work as aliases).
- **Python import names are stable**: `import process_bigraph`, `import bigraph_schema` — the rebrand did not touch them.
- **Packages**: new workspaces use a `viva_<pkg>/` package; older ones use `pbg_<pkg>/`. The Claude Code plugin is still distributed as `pbg-superpowers` (import `viva_superpowers`, with a `pbg_superpowers` shim).
- **The runtime control directory is still `.pbg/`** — that path was not renamed.
- **The scaffold repo** is `viva-template` (formerly `pbg-template`); both names resolve.

When in doubt, prefer the **viva** spelling; this guide flags the exceptions where they matter. See The stack.

Next: Home

Module catalog

Every module here is a ready-to-install unit of capability — a simulator wrapped as process-bigraph Processes, a composite, or a whole workspace — from the Vivarium Collective. Install one into a workspace with `/viva-catalog` (`install <name>`), or browse the source on GitHub.

Filter the catalog

Type to search, or click capability tags to narrow the list. There are **27 modules** across **22 capability tags**.

`spatio-flux`

GitHub ↗

spatio-temporal microbial simulations with Vivarium 2.0

`microbial` `metabolism`

```
/viva-catalog install spatio-flux
```

`v2ecoli`

GitHub ↗

Whole-cell E. coli model in Vivarium 2

`whole-cell` `ecoli`

```
/viva-catalog install v2ecoli
```

viva-amici

GitHub ↗

Process-bigraph wrapper for AMICI (CVODES-based ODE/DAE simulator). Real bridge to https://github.com/AMICI-dev/AMICI.

ode

```
/viva-catalog install pbg-amici
```

viva-biomodels

GitHub ↗

Process-bigraph workspace: BIOMODELS regression — installs pbg-biomodels-bundle and drives investigations against the BioModels corpus

```
/viva-catalog install pbg-biomodels
```

viva-biomodels-bundle

GitHub ↗

Regression harness for BioModels: runs each model under multiple simulators (COPASI, Tellurium) and renders an interactive comparison report.

```
/viva-catalog install pbg-biomodels-bundle
```

viva-bioreactordesign

GitHub ↗

Process-bigraph wrapper for BioReactorDesign (BiRD) bioreactor simulation

bioreactor

```
/viva-catalog install pbg-bioreactordesign
```

viva-caspule

GitHub ↗

Process-bigraph wrapper for CASPULE (a LAMMPS variant with dynamic bond formation/breaking)

`lammps`

```
/viva-catalog install pbg-caspule
```

viva-comets

GitHub ↗

Process-bigraph wrapper for COMETS (dynamic FBA + 2D spatial microbial ecosystems)

`microbial` `spatial` `fba`

```
/viva-catalog install pbg-comets
```

viva-composite-nfsim-caspule

GitHub ↗

Process-bigraph composite: CASPULE bond-aware MD coupled to NFSim rule-based kinetics through a configurable observable detector that converts spatial bond clusters into non-spatial species counts.

`spatial` `rule-based` `kinetics` `md` `composite`

```
/viva-catalog install pbg-composite-nfsim-caspule
```

viva-compucell3d

GitHub ↗

Process-bigraph wrapper for the CompuCell3D multicellular simulation environment

`multicellular`

```
/viva-catalog install pbg-compucell3d
```

viva-copasi

GitHub ↗

process-bigraph-compatible COPASI Steps and Processes for SBML simulation

`sbml`

```
/viva-catalog install pbg-copasi
```

viva-emitters

GitHub ↗

Focused emitter library for process-bigraph composites

```
/viva-catalog install viva-emitters
```

viva-idynomics2

GitHub ↗

Process-bigraph wrapper for IDynoMiCS 2.0, the Kreft lab's Java-based individual-based biofilm simulator

```
/viva-catalog install pbg-idynomics2
```

viva-lammps

GitHub ↗

Process-bigraph wrapper for the LAMMPS molecular dynamics simulator

`lammps` `md`

```
/viva-catalog install pbg-lammps
```

viva-martini

GitHub ↗

Process-bigraph wrapper for the Martini coarse-grained force field — membrane systems, micelles, protein-lipid complexes, and vesicles

membrane coarse-grained

```
/viva-catalog install pbg-martini
```

viva-medyan

GitHub ↗

Process-bigraph wrapper for the MEDYAN cytoskeleton simulator: pure-Python re-implementation + a subprocess-driven bridge to the real MEDYAN C++ binary (with checkpoint-restart and HDF5 vesicle support).

cytoskeleton membrane

```
/viva-catalog install pbg-medyan
```

viva-mem3dg

GitHub ↗

Process-bigraph wrapper for Mem3DG membrane mechanics simulator

membrane

```
/viva-catalog install pbg-mem3dg
```

viva-membrane-actin-composite

GitHub ↗

Process-bigraph composite: pbg-mem3dg + pbg-readdy as a Brownian ratchet (actin pushing on membrane, closed-loop)

membrane composite particle

```
/viva-catalog install pbg-membrane-actin-composite
```

viva-munk

GitHub ↗

No description yet.

```
/viva-catalog install Viva-munk
```

viva-nfsim

GitHub ↗

A process-bigraph wrapper for BioNetGen/NFSim

`rule-based`

```
/viva-catalog install pbg-nfsim
```

viva-oxidizeme

GitHub ↗

Process-bigraph wrapper for OxidizeME — a genome-scale ME-model of E. coli with ROS damage/repair (Yang et al. 2019 PNAS / Palsson lab)

```
/viva-catalog install pbg-oxidizeme
```

viva-reactive-system

GitHub ↗

Process-bigraph wrapper for Milner-style Bigraphical Reactive Systems, with a worked MAPK signalling example.

```
/viva-catalog install pbg-reactive-system
```

viva-readdy

GitHub ↗

Process-bigraph wrapper for the ReaDDy particle-based reaction-diffusion simulator

particle

reaction-diffusion

```
/viva-catalog install pbg-readdy
```

viva-simbio

GitHub ↗

Process-bigraph wrapper for simbio (Chemical Reaction Network simulation) — load models from Antimony, simulate with simbio

sbml

```
/viva-catalog install pbg-simbio
```

viva-smoldyn

GitHub ↗

Process-bigraph wrapper for the Smoldyn particle-based spatial stochastic simulator

spatial particle

stochastic

```
/viva-catalog install pbg-smoldyn
```

viva-tellurium

GitHub ↗

Process-bigraph wrapper for Tellurium / libroadrunner SBML & Antimony simulation

sbml

```
/viva-catalog install pbg-tellurium
```

viva-vcell-fvsolver

GitHub ↗

process-bigraph wrapper around pyvcell / pyvcell-fvsolver — VCell's finite-volume 3D reaction-diffusion PDE solver as a PBG Process

reaction-diffusion pde

```
/viva-catalog install pbg-vcell-fvsolver
```