.

Supplementary Materials for Vivarium: an interface and engine for integrative multiscale modeling in computational biology

Eran Agmon [1*], Ryan K. Spangler [1], Christopher J. Skalnik [1], William Poole [2], Shayn M. Peirce [3], Jerry H. Morrison [1], and Markus W. Covert [1*]

[1]Department of Bioengineering, Stanford University, Stanford, CA, USA
[2]Computation and Neural Systems, California Institute of Technology, Pasadena, CA, USA
[3]Department of Biomedical Engineering, University of Virginia, Charlottesville, VA, USA

December 13, 2021

Contents

1   Introduction

This supplement to "Vivarium: an interface and engine for integrative multiscale modeling in computational biology" includes detailed examples, additional descriptions of libraries used in the main text, tables with key terms and Vivarium methods, and coding examples. Section 2 goes into detail as it builds an example system, starting with a deterministic model of unregulated gene expression, and then adding complexity through stochastic multi-time stepping, division, and hierarchical embedding in a shared environment. This example is built up incrementally, highlighting key features of the methodology that enable incremental construction of complex models. Section 3 introduces the individual libraries used to build the multi-paradigm composite in the main text. This includes the vivarium-cobra, vivarium-bioscrape, and vivarium-multibody libraries. The appendices include tools for debugging, performance profiling of different models, references for software availability with urls, tables with key terms and methods, and code examples for building Vivarium models.

2   Interface basics

This section introduces the elements and methodology of Vivarium by working through an example system of unregulated gene expression. We provide supplementary Jupyter notebooks (links are provided in supplementary materials 5.1) that implement each of the examples in executable code – technical readers are encouraged to have these notebooks open as they read through this section.

    The guiding example separates gene expression into two processes: transcription – in which a gene is transcribed to form mRNA, and translation – in which the mRNA is translated to form a protein. We start by representing each of these functions as simply as possible

Table S2: Elements of the Vivarium framework. Some of these elements have software analogs, which are referenced in monospaced `code` format.

Term	Definition
Process	A modular sub-model that encodes a biological mechanism and can be composed with other processes to create a larger composite model. Particular process instances are created as subclasses of the `Process` class.
Store	A collection of state variables read by the processes, which contains methods for applying updates. State variables in stores that are shared by multiple processes are the only means of communication between those processes. A `Store` instance can be automatically constructed based on the processes' declared ports, variables, and their schema.
Port	A named connector on a process that gets connected to a store. Processes can declare one or more port, and the state variables they want to receive through these ports.
Schema	A state variable's declared data type, default value, and methods such as updaters and dividers, by which updates to the variable are handled. Schemas are declared by the processes' and are used to initialize stores at the start of a simulation.
Topology	Short for "process-store interaction topology", this is a bipartite network that declares how to connect processes to stores. It is declared as a Python dictionary for each process, with port names mapped to paths where stores are expected.
Composite	An integrated model with multiple processes whose connections to stores are specified by a topology. A `Composite` class has processes and a topology; it is passed to the engine to create the required stores.
Composer	A composer (subclass of `Composer`) generates composites by initializing a processes and specifying a topology for how they are wired together.
Step	`Step` is a subclass of `Process`. Step instances runs after the other processes and calculate additional state values from other available state variables – for example, concentrations from molecular counts. These are used to offload complexity from the dynamical processes.
Flow	A directed acyclic graph (DAG) – a type of workflow that declares the dependencies between Steps. A flow run is triggered between time steps, running the steps in the order determined by their dependencies. (See Listing S4)
Compartment	A store that contains inner stores and processes. Processes can connect to other compartments through boundary stores.
Hierarchy	A hierarchical network of nested stores. A hierarchy can be updated during simulation runtime by the processes with update methods such as divide, move, and add.
Engine	`Engine` is a discrete-event simulation engine. It takes processes, steps, topology, and flow as input arguments, connects them to a store hierarchy, and runs the integrated model forward in time.

with difference equations, and run them on their own. We then integrate them in a composite model and simulate them together. Next, we replace the deterministic transcription process with a stochastic process, and run a hybrid deterministic/stochastic simulation of gene expression that includes variable time steps. Finally, processes for cell growth and division are added, which allow the system to split into many separate agents that run in parallel in the simulation.

2.1   Transcription process

The transcription process used here is called "Tx", and models mRNA synthesis from DNA. We define a system with a single mRNA species, C, transcribed from a single gene, G. The chemical reaction network (CRN) – which specifies reactants, products, and a set of reactions – takes the form:

$$G \xrightarrow{k_{tx}} G + C,$$

$$C \xrightarrow{k_{deg}} \emptyset.$$
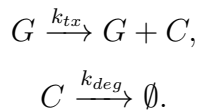
This CRN can be simulated with the difference equation

$$\Delta C = (k_{tx}G - k_{deg}C)\Delta t,$$

with C expressed from G at rate k_{tx}, and degraded at rate k_{deg}. Quantities are in concentrations (mg/mL) – this comes in useful later when converting to concentrations from counts. For pedagogical reasons, this model ignores gene copy number, RNA polymerase abundance, strength of gene promoters, and availability of nucleotides – features that could potentially be added later to improve the model's realism.

2.1.1   Ports.

An illustration of Tx is shown in Fig S2a. G and C are read through different ports, "DNA" and "mRNA", which are connected to two different stores, also called "DNA" and "mRNA". By default, a process's ports connect to stores that have the same name – the next subsection demonstrates more complex mappings. For a small model with only two variables splitting the variables into separate ports might seem excessive, but for larger models this is a useful design principle. Generally, port design should be used to organize variables by useful categories: locations such as cytoplasm, membrane, chromosome; molecule groups such as metabolites, proteins, chromosomes; or by other categories such as global variables, fluxes, concentrations.

Listing S1: Python implementation of the minimal transcription process, Tx. This demonstrates the Vivarium process interface. Defaults correspond to default parameter values which can be overwritten in the `Process` constructor. Unit conversions are supported by the pint library [7]. States are Python dictionaries which encode the file structure of a Vivarium model. Updates are similarly returned as in the same hierarchical dictionary format.

```python
class Tx(Process):
    defaults = {
        'ktx': 1e-2,
        'kdeg': 1e-3}

    def ports_schema(self):
        return {
            'mRNA': {
                'C': {
                    '_default': 100 * units.mg/units.mL,
                    '_updater': 'accumulate',
                    '_emit': True,
                }},
            'DNA': {
                'G': {
                    '_default': 10 * units.mg/units.mL,
                }}}

    def next_update(self, time_step, state):
        # Retrieve the state variables through the ports
        G = state['DNA']['G']
        C = state['mRNA']['C']

        # Run the model
        dC = (self.parameters['ktx']*G - self.parameters['kdeg']*C) * time_step

        # Return an update
        return {
            'mRNA': {
                'C': dC,
            }
        }
```

2.1.2 Process interface.

Listing S1 shows Python code for the Tx process – an instance of the `Process` class. Making a process requires implementing the process interface, which involves the following constant and methods: 1) `defaults`: This class constant declares expected parameter names and values – even if only with empty values that get replaced upon initialization. The process constructor (`__init__`) accepts a list of parameters when a process is initialized, which override the defaults. 2) `ports_schema`: This method declares a process' ports ("RNA" and "DNA"), the variables that are accessed through those ports (C and G), and their required schemas (Supplementary materials, Table S9). Any type of value can be used in the
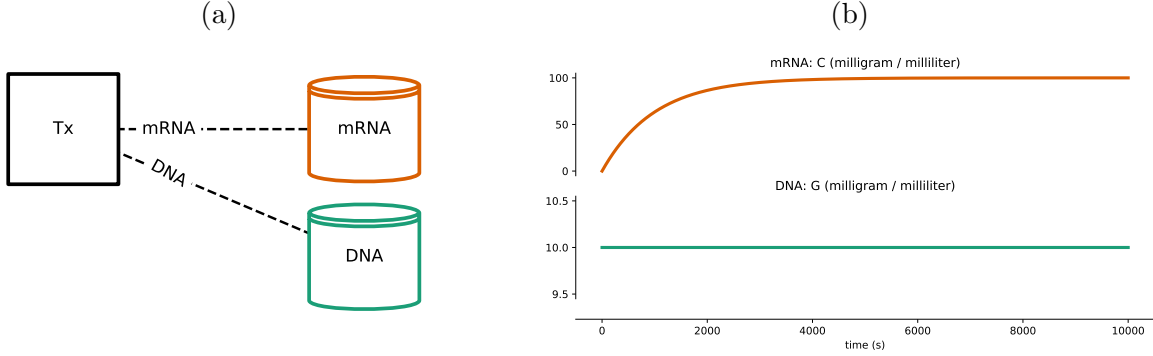
Figure S2: Tx – a transcription process. (**a**) The system's topology, with process Tx wired to two stores – DNA and mRNA – through ports of the same name. (**b**) Simulation output. The DNA G remains fixed at its initial value, and the mRNA C increases up to a steady state.

schema, such as integers, arrays, or more complex data structures. For novel data types, new schema methods such as updaters and dividers might be required – these are modular and can be defined by users. Updaters and dividers available with vivarium-core are listed in (Supplementary materials, Tables S10 and S11). 3) `next_update()`: This method contains the dynamical model. The steps of this method involves retrieving the variables through the ports, applying the encoded mechanism for the time step's duration, and returning the update for each port.

2.1.3 Simulating a process.

The output of Tx is shown in Fig S2b. Individual processes can be run on their own by the simulation engine, which initializes the stores, runs the simulation, and saves the output. Each simulation is configured with an emitter, which logs the state of variables marked to emit during runtime – marking a variable to emit can be declared in the process' `port_schema`. If the emitter is connected a database (we use mongoDB), the saved data can be retrieved from the database for visualization and analysis at any time during or after a simulation run.

2.2 Transcription/Translation composite

Next, we integrate the Tx transcription process with a translation process called "Tl", whose implementation is not shown here but is available in the supplementary notebook. Tl takes a similar form to Tx, but with protein X translated from mRNA C and degraded:

$$\Delta X = (k_{tl}C - k_{deg,X}X)\Delta t.$$

As before, units are in concentrations (mg/mL). As a simplifying assumption, translation rate considers ribosome availability, strength of ribosome binding to mRNA, availability of tRNAs and free amino acids to all be part of one lumped constant.

Fig S3a illustrates the composite model called "TxTl", with Tx and Tl both wired to a shared store called mRNA. This couples the two processes, so that mRNAs synthesized by Tx impacts the expression of proteins by Tl.

2.2.1 Composer interface.

`Composer` is a class that generates composites. A given composer's inherited `generate()` method calls `generate_processes()` to construct the processes, `generate_steps()` to construct the steps, `generate_topology()` to wire the processes to the stores, `generate_flow()` to declare the dependencies between steps. When called, it returns a composite that is ready for simulation. Making a composer involves the following class attributes: 1) Composers have their own `defaults` for parameters, which can override the default parameters for individual processes, thus providing easy access for parameter scans and learning algorithms to adjust the full composite's behavior. 2) The `generate_processes()` method constructs a composite's processes in a dictionary, which maps each of their names to the instantiated process objects. 3) The `generate_steps()` method constructs a composite's steps in a dictionary, which maps each of their names to the instantiated step objects – steps declared in generate_processes() are automatically detected and used with a sequential dependencies based on the order they are declared in the dictionary. 4) The `generate_topology()` method returns a topology dictionary that declares how each process's ports connect to stores in the hierarchy. 5) The `generate_flow()` method returns a topology dictionary that declares each step's dependencies on other steps (see Listing S4).

The TxTl composite is specified in listing S2. Default parameters are empty so the processes will use their own defaults if none are supplied. Each process is constructed in `generate_processes()`, and wired together in `generate_topology()`. Tx's DNA port maps to a store called "DNA", Tl's protein port maps to a store called "Protein", and both Tx and Tl get wired to the same "mRNA" store containing the state variable "C", thus coupling the two processes together.

2.2.2 Using the Vivarium engine

The vivarium engine runs TxTl to produce the simulation output shown in Fig S3b, where the mRNA reaches a steady-state as before while the protein concentration increases over time following a logistic-like curve.

Composite models are run with vivarium-core's `Engine` class. It is a discrete-event simulation engine that can accept either processes and a topology (the attributes of a `Composite` class) and generates the store hierarchy, or a pre-generated `Store` with the hierarchy already made. Some additional keyword arguments to engine let you pass in an initial state, emitter time steps, experiment name, and emitter type (for example print to console, save to RAM, save to database). Listing S3 shows how to take a composer, generate a composite, pass its processes and topology to the engine, and run a simulation.

Listing S2: Python code for the TxTl Composer. This demonstrates the composer constants and interface methods. Processes are initialized in `generate_processes` and their ports are wired to stores by declaring a topology in `generate_topology`. Listing S7 in the Supplementary materials includes more advanced examples of `generate_topology`, including connecting ports to stores at different levels of the hierarchy, splitting a port into multiple stores, and connecting variables with different names.

```python
class TxTl(Composer):
    defaults = {
        'Tx': {},
        'Tl': {}}

    def generate_processes(self, config):
        return {
            'Tx': Tx(config['Tx']),
            'Tl': Tl(config['Tl'])}

    def generate_topology(self, config):
        return {
            'Tx': {
                'DNA': 'DNA',
                'mRNA': 'mRNA'},
            'Tl': {
                'mRNA': 'mRNA',
                'Protein': 'Protein'}}
```

2.3   Adding complexity with a stochastic process

Importantly, Vivarium enables users to compare competing models of a given process, simply by exchanging one for the other and simulating the resulting behaviors. The interface introduced above makes it easier to define and integrate process modules – now we demonstrate how changes in the sub-models, building off prior model design, can provide additional model functionality.

We begin by replacing the deterministic transcription process, Tx, with a stochastic process called "stochastic Tx". The biological reasoning for this might be that in individual cells many genes are transcribed at low expression rates and synthesize small counts of mRNA, which leads to stochastic behavior. We use the Gillespie algorithm [4] – a discrete and stochastic method for systems with few reactants – to simulate individual reactions. The Gillespie algorithm can be broken into two steps – one for calculating the time which elapses before an event occurs, and the other for determining the nature of that event. Stochastic simulations require variable time steps; for example, the distribution of time steps in a simulation using stochastic Tx is shown in Fig S4c.

2.3.1   Steps.

The Gillespie algorithm operates on molecular counts – every reaction event increases the counts of the products and decreases the counts of the substrates. Thus, the model needs to convert the molecular counts from the stochastic Tx process to concentrations for input to
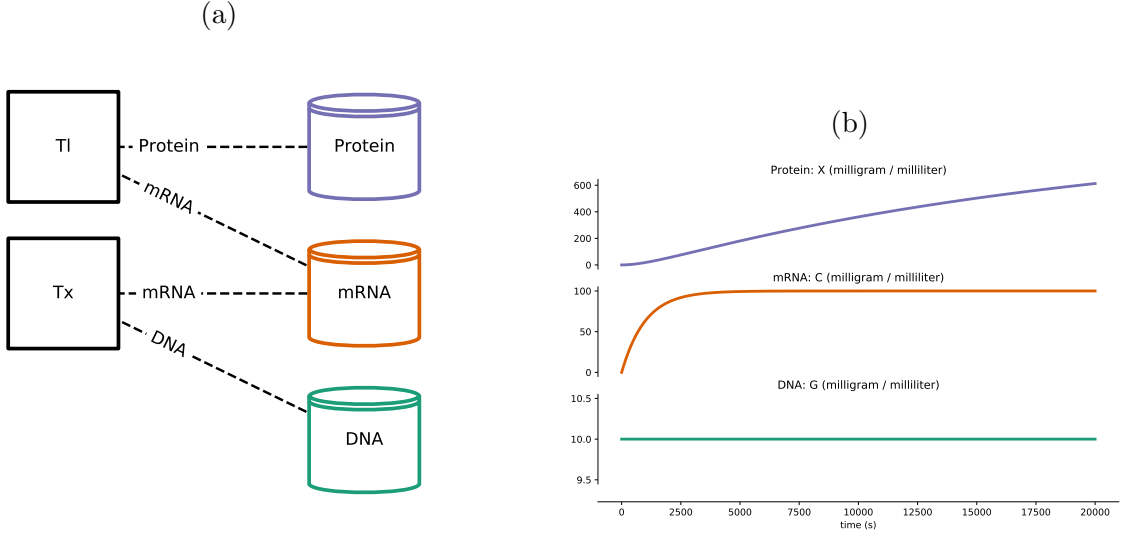
(a)

(b)

Figure S3: The TxTl composite with Tx transcription and Tl translation. (**a**) The model's topology and process declaration. Tx is the same as in Fig S2, and Tl is a new process with ports for mRNA and protein. Both processes are wired to the same mRNA store, thus coupling them together. (**b**) The simulation output of this model is the same as in Fig S2 showing DNA G and mRNA C, but with added protein X also being expressed.

the Tl process. To perform this conversion, we add an auxiliary *step* process.

`Step` is a subclass of `Process`, but without a time components. These instances run after the dynamic processes, and in a dependency graph called a *flow* (like a workflow). For example, if there were processes A and B with dt=1, C with dt=2, and two steps D1 and D2 with D2 depending on D1, the run order would be something like:

$$t_0 : D1, D2$$

$$t_0 \rightarrow t_1 : A, B, D1, D2$$

$$t_1 \rightarrow t_2 : A, B, C, D1, D2$$

There are many uses for steps. These could act as translators that adapt different modeling formats, lift and restriction operators that translate between scales, and helper processes to offload complexity.

The vivarium-core library provides several general-purpose steps. In the current example we instantiate a step called "counts to mg/mL" in Fig S4 – this step calculates new concentrations from counts after every step.

2.3.2 Multiple timescales.

For processes to operate at different timescales, the simulation engine handles updates on a per-process basis. At the start of each process' time step, the engine retrieves the process' required time step by calling its `calculate_timestep` method with the current state of the system. By default, the process returns a fixed time step that can be declared in their parameters, but stochastic Tx calculates a new time step by using the Gillespie algorithm.

Listing S3: Running a composite with `Engine`. This listing demonstrates how to initialize a composer by passing in parameters, how to generate a composite using a composer, and how to pass a composite's processes and topology into the simulation engine. The engine is then run for a length `total_time`, and its output data is retrieved from the emitter.

```python
# define configuration, to override default parameters
config = {
    'Tx': {'ktsc': 1e-2},
    'Tl': {'ktrl': 1e-3}}

# initialize TxTl composer with the config
composer = TxTl(config)

# generate the composite, within initialized processes
composite = composer.generate()

# define an initial state, to override default values
initial_state = {
    'DNA': {'G': 10.0 * units.mg / units.mL},
    'mRNA': {'C': 0.0 * units.mg / units.mL},
    'Protein': {'X': 0.0 * units.mg / units.mL}}

# initialize a simulation with the composite's processes and topology
experiment = Engine(
  processes=composite.processes,
  topology=composite.topology,
  initial_state=initial_state)

# run the experiment
total_time = 20000
experiment.update(total_time)

# get the simulation output
output = experiment.emitter.get_data()
```

After retrieving the time step, the engine calls the process' `next_update` method with the current state of the system. When the system time reaches the end of the process' time step, it retrieves the update and applies it to the system state. This way, all processes can run at their preferred timescales. With the current version of the engine, the user needs to make sure the time steps of the processes are synchronized with each other to avoid numerical issues. Future versions can introduce a specialized adaptor process to handle the processes' time steps in a way that automatically ensures coordination.

2.4   Hierarchical embedding

Up to this point, each model had a fixed number of processes and stores, with a fixed topology. In contrast, cell division requires a hierarchy with agents embedded in a shared environment, within which the cell agents can grow and divide. The hierarchy needs to launch new processes and stores for each agent created during runtime.
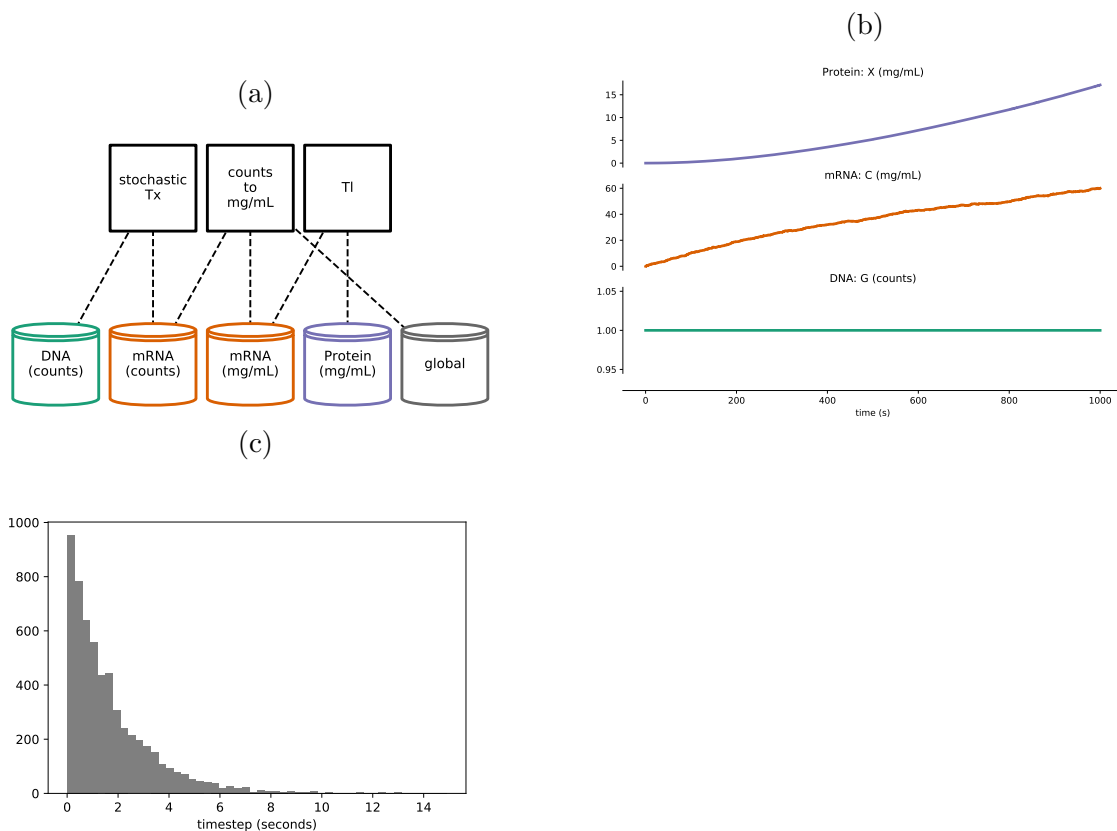
Figure S4: Stochastic transcription with deterministic translation. The stochastic process adjusts its time step based on the total propensity of the system, which is recalculated at every step. (**a**) The topology, showing the stochastic Tx process, the counts to mg/mL concentration step, and the deterministic Tl process. Counts to mg/mL is connected to a "global" store, which holds the volume variable required to calculate concentrations. (**b**) The simulation output shows stochastic dynamics of mRNA, and the impact of this stochasticity on the protein concentration. (**c**) Histogram shows the variable time steps resulting from running stochastic Tx on its own for $10,000$ seconds. Tl on its own runs at fixed 1 second intervals.

Agents and environments. When one compartment is nested in another, the inner compartment can be considered an agent and the outer compartment its environment. Coupling between an agent and an environment is supported by their processes sharing variables in boundary stores. For example, agent processes can update boundary variables required by environmental process such as agent volume, shape, motile forces, and uptake of molecules. Environmental processes can update the boundary conditions of agents' internal processes; for example, local molecular concentrations, and temperature. In the current example, a process called "colony volume" is added to the environment to calculate the volume of all the agents together (Fig S5a). This derived population-level state variable could in principle be used to drive other mechanisms – for example if simulated in a gut microenvironment, bacterial colony volume variable could be used to impact the host's digestion.

Advanced process-store topology. Up to this point, the system comprised a single

11

compartment (i.e., a single cell). Accordingly, each topology specified a simple path from each port to a store in the same compartment – a "flat" network. Now, we model the environment as an outer compartment which can contain one or more cells. This requires advanced specifications in a composer's `generate_topology` that connect a processes' ports to stores further up or down the hierarchy. A port connects to a store in the hierarchy by specifying a path, which could go up the hierarchy to stores in outer compartments; or down the hierarchy to stores contained in inner compartments. `generate_topology` also supports splitting ports to draw from variables in separate stores, merging ports to draw from the same store, and aliasing names to variables to connect models with different variable names. A few technical examples of these advanced topology methods are included in Supplementary materials section 5.6.

In the current example, the colony volume process reads the counts of all molecules through an "agents" store, which contains all of the individual agent instances – each with its own DNA, mRNA, and Protein, and global stores. Colony volume is configured to read the volumes in the individual global stores, and calculates total colony volume.

Division. To enable a compartment to divide during runtime, Vivarium provides a division process that is configured with a divide condition, which when true triggers division. A configurable condition means the process could be reused for more sophisticated cell models, for example based on the completion of chromosome segregation or the formation of a septum. Upon division, the mother's variables' states are divided between daughter agents based on those variables' 'divider' schema methods (Table S11).

For our example system, we initialize agents at 1000 fg, and trigger division when they double that mass. Fig S5c shows how the output over four generations of growth and division, starting off with a single stochastic TxTl instance that splits into two independent instances and then four and then eight – each of which exhibits its own distinctive behavior.

Compartment hierarchy updates. In biological systems, the nesting of compartments can be rearranged over time with behaviors such as engulfing, merging, and division. To support these behaviors, hierarchies in Vivarium can also be restructured during runtime. There are several built-in hierarchy update methods including move, add, delete, divide, and generate. Processes can trigger these in different combinations to generate a wide range of possible behaviors including merge, two neighboring compartments combine into one; burst, a compartment combines with its environment; engulf, one compartment is moved inside of a neighboring compartment; and expel, a compartment moves to be a neighbor of its outer compartment. All of these are available in vivarium-core.

Fig S5a shows the hierarchy after one division event, with two agents embedded in the top-level "agents" store. This requires the simulation to instantiate new agents – remove the mother agent, and generate two daughter agents with the mother's composer, and an inherited state.

(a)

(b)

```
# make txtl composite with agent id "0"
agent_id = '0'
txtl_composer = TxTlDivision(agent_config)
agent0 = txtl_composer.generate(
    {'agents': agent_id})

# make a colony composite
colony_composer = ColonyVolume()
colony_composite = colony_composer.generate()

# perform merge
# agent0 in store "0", under "agents"
colony_composite.merge(
    composite=agent0,
    path=('agents', agent_id))
```
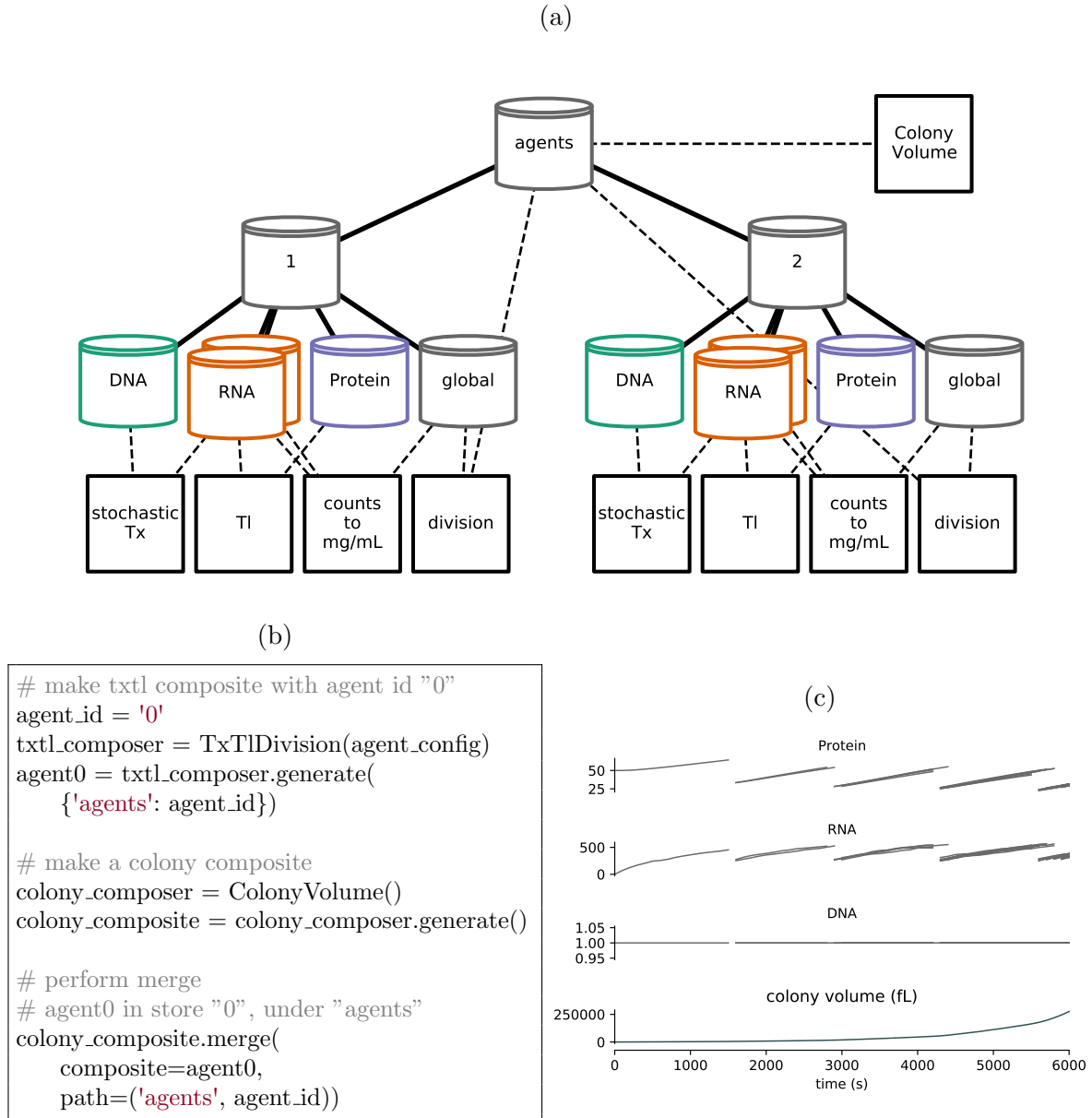
(c)

Figure S5: Hierarchical embedding and division. **(a)** Topology plot of the embedded hierarchy, with two cell agents. This plot was generated after one division event occurred, to illustrate how embedding works in Vivarium. Solid edges reflect hierarchy's directory structure, with "agents" at the top level. Dashed edges are topology connections between processes and the stores. **(b)** A script for making a composite model from a TxTl agent and a colony-level process to measure total colony volume. Agent 0 agent is placed within an 'agents' store, while Colony Volume is placed at the top level. The simplicity of the merge operation provides a flexible API for combining different models. **(c)** Simulation output of model over multiple generations of growth and division. Many instances run in parallel by the end of the simulation. The bottom plot shows the total colony volume, calculated from the full set of cell models within the environmental compartment.

13

3 Individual paradigms of Section 3

The main text demonstrates the integrative capabilities of of Vivarium by building a large multi-paradigm composite model of glucose-lactose diauxie in a colony of growing and dividing *E. coli* agents. The individual processes were built with wrappers around existing popular modeling libraries, which were imported into the glucose-lactose diauxie project and wired together into the large integrative model.

The individual processes are available for re-use in the Vivarium Collective: vivarium-cobra, vivarium-bioscrape, and vivarium-multibody. The PIPy versions used for this paper are vivarium-cobra==0.0.18, vivarium-bioscrape==0.0.0.7, vivarium-multibody==0.0.13. This section describes each of these libraries individually, shows their processes run on their own, and provides the parameters used to configure the example in Section 3.

3.1 Flux-balance analysis with COBRA

Flux balance analysis (FBA) is an optimization-based metabolic modeling approach that takes network reconstructions of biochemical systems, represented as a matrix of stoichiometric coefficients and a set of flux constraints, and applies linear programming to determine flux distributions, for example those that maximize the production of biomass based on the known composition of metabolic end-products [6]. A strength of FBA is its capacity to simulate whole-network flux distributions using a minimal set of parameters. FBA is made dynamic (called dFBA) by iteratively re-optimizing the objective with updated constraints at every time step [11]; these constraints change with environmental nutrient availability, gene regulation, or enzyme kinetics. Many useful tools related to building and simulating FBA models have been developed and made freely available in the COBRA toolbox, which is also available in python as COBRApy [2].

For this work, we developed a Vivarium process that provides a wrapper around CO-BRApy, and is located in the vivarium-cobra library (Fig S6a). This process, called "CO-BRA", can be initialized with a BiGG model from the BiGG model database [5]. BiGG models are genome-scale metabolic models, which are available for dozens of *E. coli* strains, as well as many other cell types. The model used here is *iAF1260b*, which includes 2382 reactions, controlled by 1261 genes, and with an objective that includes the production of 67 molecules.

For purposes of integration with Vivarium, we pass the COBRApy results into internal metabolite pools that are available for other processes to utilize. The COBRA process includes a "flux bounds" port, which allows other processes to dynamically modify the flux constraints on the FBA problem. Accordingly, some additional processes were developed so that the COBRA process could support dFBA (shown in Fig S6a). These processes include "local field" to model the external environment with dynamic molecular concentrations, and "mass step" to convert the internal metabolite counts into a total mass.

Thus, when COBRA is run as a composite with the local field and mass step processes (Fig S6b), it takes up metabolites from the environment and grows its internal pools of metabolites, exponentially increasing in mass and reproducing the expected 40 minute doubling time in minimal glucose media.
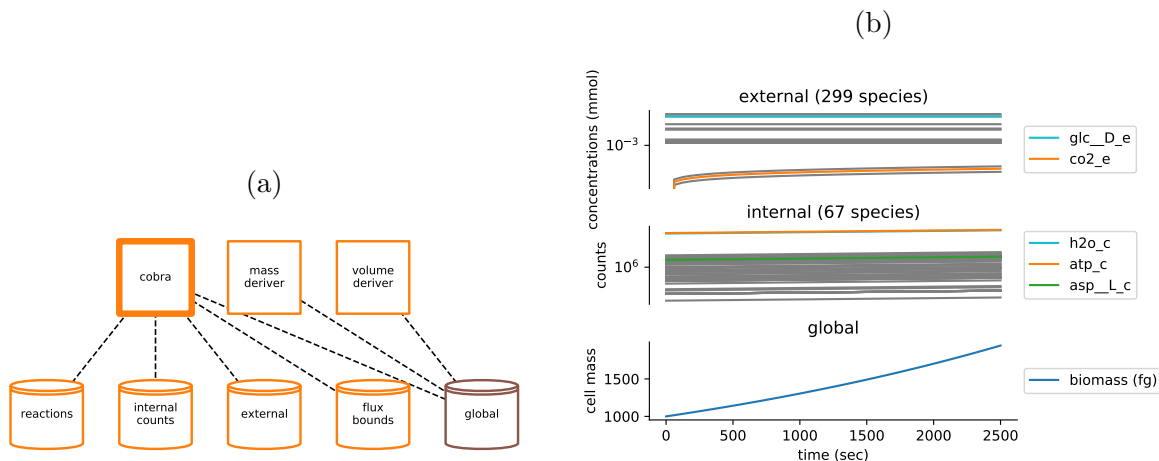
Figure S6: Demonstration of the COBRA composite's dynamic FBA. **(a)** Topology of the COBRA composite, which models metabolism with the COBRA process, and additional processes "local field" and "mass step" and help make the system into a dynamic FBA with an external environment, and total compartment mass based on individual metabolite pools and their molecular weights. The COBRA process has ports connected to "reactions", "internal counts", "external" (which are in concentrations), "flux bounds", and "global". **(b)** Simulation of a COBRA composite model, configured with BiGG model *iAF1260b*. Environmental concentrations (top) and internal molecule counts (middle) are plotted in log-scale due to the wide range across molecular species. The internal metabolites are multiplied by their molecular weight and summed to get total biomass (bottom).

COBRA process		
Parameter	Value	Description
BiGG model	$iAF1260b$	metabolic model passed into the COBRA process, from [3]
time_step	$60s$	fixed process time step

Table S3: Processes parameters for vivarium-cobra version 0.0.18.

3.2 Chemical reaction networks with Bioscrape

To add a CRN network model of transcription, translation, regulation, and the enzymatic activity of the lac operon and its resulting proteins, we turned to a published model [9], with many parameters from [12]. Our model includes all the same features, except for the addition of different combinatorial conformations of the lacR repressor binding to the lac operon. The model includes 11 species and 10 reactions, using a combination of mass action and Hill function propensities to represent the following functions: production of mRNA from the Lac Operon, translation of the proteins, tetramerization of β-Gal, lactose degradation to allolactose, hydrolysis to glucose and galactose, import of glucose and lactose, degradation of mRNA and proteins, and dilution for the deterministic version of the model. We converted this model to an SBML format using BioCRNpyler – an open source tool for specifying CRNs [8]. With the model in SBML, we were ran simulations with a Vivarium process

built with Bioscrape (Bio-circuit Stochastic Single-cell Reaction Analysis and Parameter Estimation) [10] – a Python package that supports deterministic and stochastic simulations. The "Bioscrape" process is available at the vivarium-bioscrape library. Its topology can be seen in Fig S7a.

Running the lac operon CRN model in isolation shows expected behavior (Fig S7b), with glucose initially being taken up from the environment while lactose is not. Once external glucose is depleted, the lac genes are expressed, concentrations of β-Galactosidase and lactose permease rise, and lactose is brought into the cell and degraded. This is all done smoothly, with continuous dynamics (Fig S7b, left). Using the Bioscrape process also facilitates a stochastic simulation of this CRN (Fig S7b, right) the results of which show lac operon RNA expressed via a randomly-occurring transcription event, followed by expression of the lac proteins, and enabling subsequent import of lactose. For the stochastic model, external nutrients can only exist in a small external environmental volumes – large environments make for large nutrient counts, which slows the stochastic simulator drastically and makes simulations unfeasible. This limitation is corrected by the integrated model, which introduces many separate external locations for nutrients.

The Bioscrape process includes ports for "species", "delta_species", "rates", and "global". All the individual molecular species (including genes, RNA, and protein) connect to the the "species" port; "delta_species" is an output of the process that holds the changes (deltas) to the species in a given timestep, "rates" are instantaneous rates determined by the simulator, and "global" reads the "volume" variable.
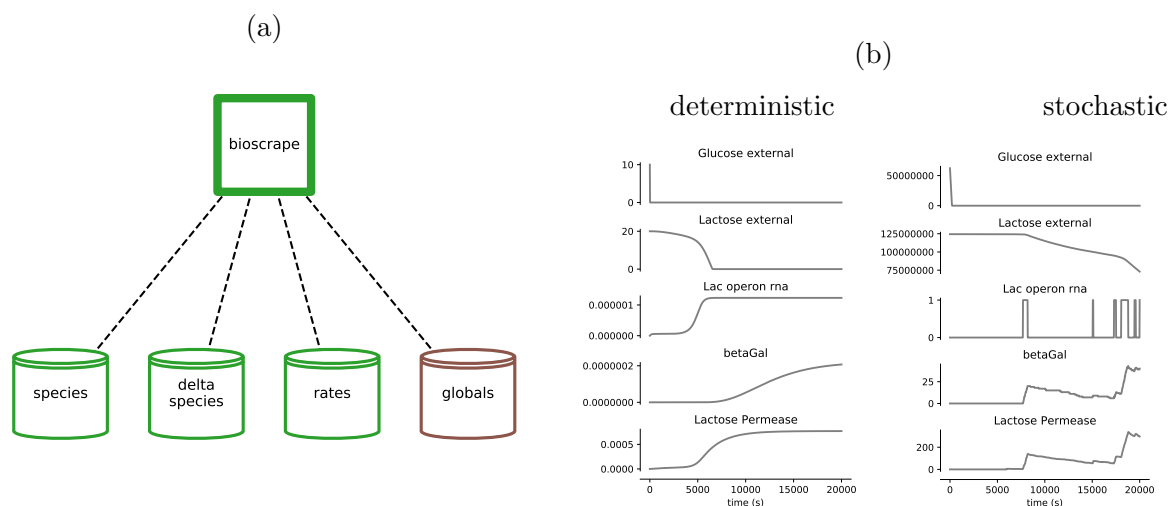

Figure S7: Demonstration of the Bioscrape process on its own. **(a)** Topology of the Bioscrape process, which models a CRN with deterministic (ODE) and stochastic (Gillespie) simulators. Bioscrape has ports connected to "species", "delta species", "rates", and "global". **(b)** Simulation of the Bioscrape process configured with a lac operon model. On the left – a deterministic simulation models the smooth dynamics of molecular concentrations. On the right – A stochastic simulation models the discrete events with molecular counts.

Bioscrape process: deterministic model		
Parameter	Value	Description
stochastic	*False*	use Bioscrape's stochastic simulator
initial_volume	1	compartment volume for the simulator
internal_dt	0.01	internal time step with which to run the simulator
time_step	10s	fixed process time step
Bioscrape process: stochastic model		
Parameter	Value	Description
stochastic	*True*	use Bioscrape's stochastic simulator
safe_mode	*True*	use Bioscrapes safe model interface
initial_volume	1	compartment volume for the simulator
internal_dt	0.01	internal time step with which to run the simulator
time_step	10s	fixed process time step

Table S4: Processes parameters for vivarium-bioscrape version 0.0.0.7.

3.3 Multicell physics with pymunk and field diffusion

With individual cells being represented by the Bioscrape and COBRA processes, our final step was to model a spatial environment in which these cells can grow, divide, and interact – through physical forces as well as by uptake and secretion of molecules in a shared chemical milieu. The environment is implemented using a composite from the vivarium-multibody library called "lattice", which consists of two processes: "multibody" and "diffusion" (Fig S8a).

The multibody process is a wrapper around the physics engine pymunk [1], which can model individual cell agents as capsule-shaped rigid bodies that can move, grow, and collide. Multibody tracks the following boundary variables for each agent: location, length, width, angle, mass, thrust, and torque. The physics engine applies these variables for the update time step, and returns a new location for each agent. Agents can update volume, mass, and motile forces, thus impacting their movement in the environment. Upon division, the `daughter_location` divider is applied to the location of agents, so that when they divide the daughters are placed end-to-end in the same orientation as the mother.

The diffusion process was developed for this study as a custom process, with no pre-existing simulator (i.e., not using pymunk). It simulates bounded two-dimensional fields of molecular concentrations. Each lattice site (x, y) holds the local concentrations of any number of molecules, and diffusion simulates how they homogenize across local sites. Each agent can uptake and secrete molecules at its position in the field. The implementation uses an adaptor process called "local field", which converts exchanged molecules from the given agent to concentrations at the agent's location.

Fig S8b shows the lattice composite simulated with minimal grow/divide agents. A single initial agent grows and divides to form colonies of many minimal agents in the environment – as they grow, they push against each other via the multibody process, and the colony increases in volume. The agents shown in this minimal simulation do not take up molecules. Therefore, in order to demonstrate the diffusion process, we initialized the system with a concentration gradient, which lessens over time.

Lac operon expression and transport kinetics		
Parameter	Value	Description
Translation and complexation of β-Galactosidase and lacP		
k_tl_beta_Gal	9.4/60	translation rate of β-Galactosidase
k_tl_lacP	18.8/60	translation rate of lacP
BGal_ tetramer- ization	1000	complexation rate of β-Galactosidase, assumed to be fast and irreversible
Degradation of lactose with proportional Hill function		
BGal_vmax	300	rate constant of lactose degradation to allolactose, catalyzed by β-Galactosidase
Bgal_Kd	1.4	dissociation constant
Import of glucose with proportional Hill function		
GluPermease_vmax	301	rate constant
GluPermease_Kd	0.015	dissociation constant
LacPermease_ re- verse_vmax	71.38/60	rate constant for reverse reaction
LacPermease_Kd	14.62	dissociation constant
Protein degradation with mass action		
kdeg_mRNA	0.47/60	mRNA degradation rate
kdeg_prot	0.01/60	protein degradation rate
k_dilution	0.02/60	dilution rate for deterministic model, corresponds to a doubling time of 30 minutes.
stochastic model parameters		
Bgal_Kd	84310	dissociation constant for stochastic model
GluPermease_Kd	9033	dissociation constant for stochastic model
LacPermease_Kd	8800000	dissociation constant for stochastic model

Table S5: Model parameters for deterministic and stochastic models of the lac operon. These are a mixture of mass action and Hill function propensities. They were used with BioCRN-pyler [8] to generate the SBML model and save '.xml' files, which are read by the Bioscrape Vivarium process. The files, LacOperon_deterministic.xml and LacOperon_stochastic.xml are available in the vivarium-notebooks repository.
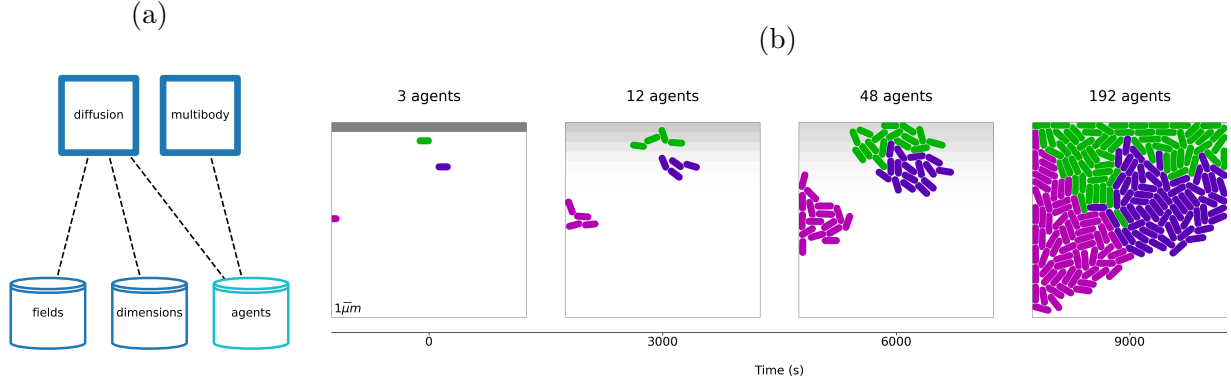
Figure S8: Demonstration of Lattice composite. **(a)** Topology of lattice composite, with a diffusion process and a multibody process, and ports connected to "fields", "dimensions", and "agents" stores. **(b)** Three grow/divide agents are initialized in the lattice. As the agents grows and divide, the multibody process simulates volume exclusion, which pushes their neighbors away and grows the colony. In this particular case, the agents do not exchange molecules with the external field, but diffusion can be seen by the spread of molecular concentrations initialized at the top row of the field.

Diffusion process		
Parameter	Value	Description
bounds	$(30\mu m, 30\mu m)$	(x, y) dimensions of the environment.
n_bins	$(30, 30)$	Number of discrete bins in the (x, y) dimensions.
depth	$0.5\mu m$	The depth of the environment. Shallow environment deplete faster.
concentrations	$glucose : 10mmol$ $lactose : 50mmol$	Initial concentrations of the environmental fields.
diffusion	$0.02\mu m^2/s$	Diffusion rate constant for all molecules.
Multibody process		
Parameter	Value	Description
bounds	$(30\mu m, 30\mu m)$	(x, y) dimensions of the environment.
jitter_force	$1 * 10^{-4} pN$	A random force applied to each agent to simulate Brownian motion.
agent_shape	$segment$	The shape of the agents. Segment shapes are like rectangles with rounded ends – a 2D capsule.

Table S6: Processes parameters for vivarium-multibody version 0.0.13.

4   Design Methodology

This section covers additional details regarding technical and conceptual design considerations. In 4.1 we look at performance tests that evaluate how Vivarium simulations scale as complexity is increased. 4.2 discusses tradeoffs in the design of Vivarium simulations.

4.1   Performance

The total runtime of a Vivarium simulation can be decomposed into two main computational costs (Fig S9a): the process update runtime, which is the cumulative time spent in the processes' `next_update` method; and second, the overhead runtime, which Vivarium requires to manage the updates from the ports, apply them to the stores, and synchronize the processes. With the parallelization of processes, total runtime can equal less than the combined process update and Vivarium overhead times, since process updates are performed concurrently. An ideal simulation would spend most of its computation in the process updates as this is both the scientifically relevant computation for the simulation, and the computation that can be easily parallelized.

To evaluate these costs, we profiled a series of simulations which varied in: 1) the number of processes, 2) the number of ports per process, 3) the number of variables updated by each process, 4) the number of parallel processes, 5) the number of CPUs available on a virtual machine (Fig S9b-c). Each simulation was evaluated by cumulative process update time, cumulative Vivarium backend time, and total simulation run time. Comparing these metrics allowed us to evaluate Vivarium's overall computational scalability.

The profiled simulations used a custom Vivarium process that generates random updates and can be reconfigured with different numbers of variables, different numbers of ports for those variables, and busy waits to precisely control how much computation happens within the update. A custom Vivarium composite assembles these processes with topologies of varied complexity. The Vivarium process, the Vivarium composite, and the profiling scans analyzed here are included in the vivarium-core repository at the address listed in 5.1. The ability to control the complexity of this process and composite allowed for more precise evaluations of Vivarium's overall scalability than the model described in Section 3 – that model used a fixed combination of certain processes (some large, some small), its division during runtime adds/removes processes, and Bioscrape has its own internal parallelization that obscures the parallelization implemented for Vivarium.

Topological complexity. The complexity of a composite's topology is related to the number of variables coming from each process and the number of ports from these processes. Figure S9b shows that the Vivarium overhead scales roughly linearly with both the number of ports and the number of variables for each processes. Linear scaling is expected, since Vivarium has to handle additional updates from the processes, and port them to different stores. In contrast, process update times are essentially unchanged by either the port or variable number.

Process complexity. Individual processes can vary in how much computation they require. In general, larger processes can be expected to take seconds to return an update, while simple processes may take microseconds and demand more frequent synchronization. Figure S9c shows three profiling scans with different wall times for the process updates. These

scans show that both process update and Vivarium overhead scale linearly with added processes. However, the relative slope of these relationships are determined by the computation performed by each process. In the top panel of Figure S9c, Vivarium overhead runtime scales more slowly than process update runtime. This indicates a simulation that is scaling well, spending relatively more time performing scientifically-meaningful computation as processes are added. In the bottom panel, the Vivarium overhead increases as more processes are added. This is an indication of poor scaling, with increased computation required to synchronize each additional process. On the computer that performed these scans, the threshold of process update scalability, where the overhead and update runtimes both increase at roughly the same rate as processes are added, was 0.08 milliseconds for each process update (Figure S9c, middle).

Parallelization. To fully utilize the available CPUs, individual processes can be run in parallel using Python's multiprocessing library. Figure S9d (top) shows 80 total processes simulated on a computer with a 8 CPUs, with n of the processes run in parallel. As n increases, the simulation time decreases, since more processes can run at the same time. This decrease levels off at around 70 parallel processes, which is the point at which there are about 10 processes for each CPU. Figure S9d (bottom) shows 50 total parallel processes simulated on different virtual machines on Google cloud, which vary by their number of CPUs. As the number of CPUs increases, total simulation time decreases, since more of the parallel processes can get their own designated CPU. This approximately levels off when the number of parallel processes equals the number of CPUs.

4.2   Process, Composite, and Library Design

Process design. The performance results suggest that there are design considerations for what makes a process optimally composable and parallelizable. If processes are too small, the overhead runtime will increase and the simulation will be poorly scaled. On the other hand, processs that are too large may have low re-usability (i.e., processes should be designed for general application and not include computations that would exclude them from many use-cases). This means that most processes should not be decomposed to their most basic computations, but rather grouped in computations of complex mechanisms that can be reused to update many molecules or other variables at a time. We believe that limitations on scalability will be primarily driven by the code most relevant to the simulation – within the processes – rather than overhead from Vivarium. That said, users should design their processes in a way that is most convenient and/or useful for them.

Composite design. Composites are bundles of processes that should work synergistically in representing particular classes of biological systems. Composites themselves may be used as modular elements, and brought together into higher-order composites of increased complexity. An example of a composite in this work is the Lattice environment, which includes both the multibody and diffusion processes. These are used together because they represent the microbial physical and chemical environment, so it is ideal to keep them bundled as a unified whole. As another example, one could imagine building a composite organelle model such as a mitochondrion, which we would want to slightly reconfigure and reuse in different Eukaryote cell models. Composites can be designed in such a way that users can easily reconfigure them by passing options to the composite constructor, without

having to delve into any process implementation or the composite's wiring. **Library design.** Vivarium-core is a stand-alone library without any significant modeling modules. The modules used for this paper's examples are in external libraries that can be brought together and wired in more arbitrary configurations. This approach has all of the advantages we have listed above and in the main text. However, it can also be difficult to maintain separate libraries and test software changes across libraries. Thus, if processes are expected to be developed in parallel as the model is built, developers should consider bringing them into the same Vivarium library.

Libraries may also be set as private or public. Users might prefer to keep their models private while they are being developed, and only make them public upon publication of the model – at that point, others could import the model and use it within separate projects.
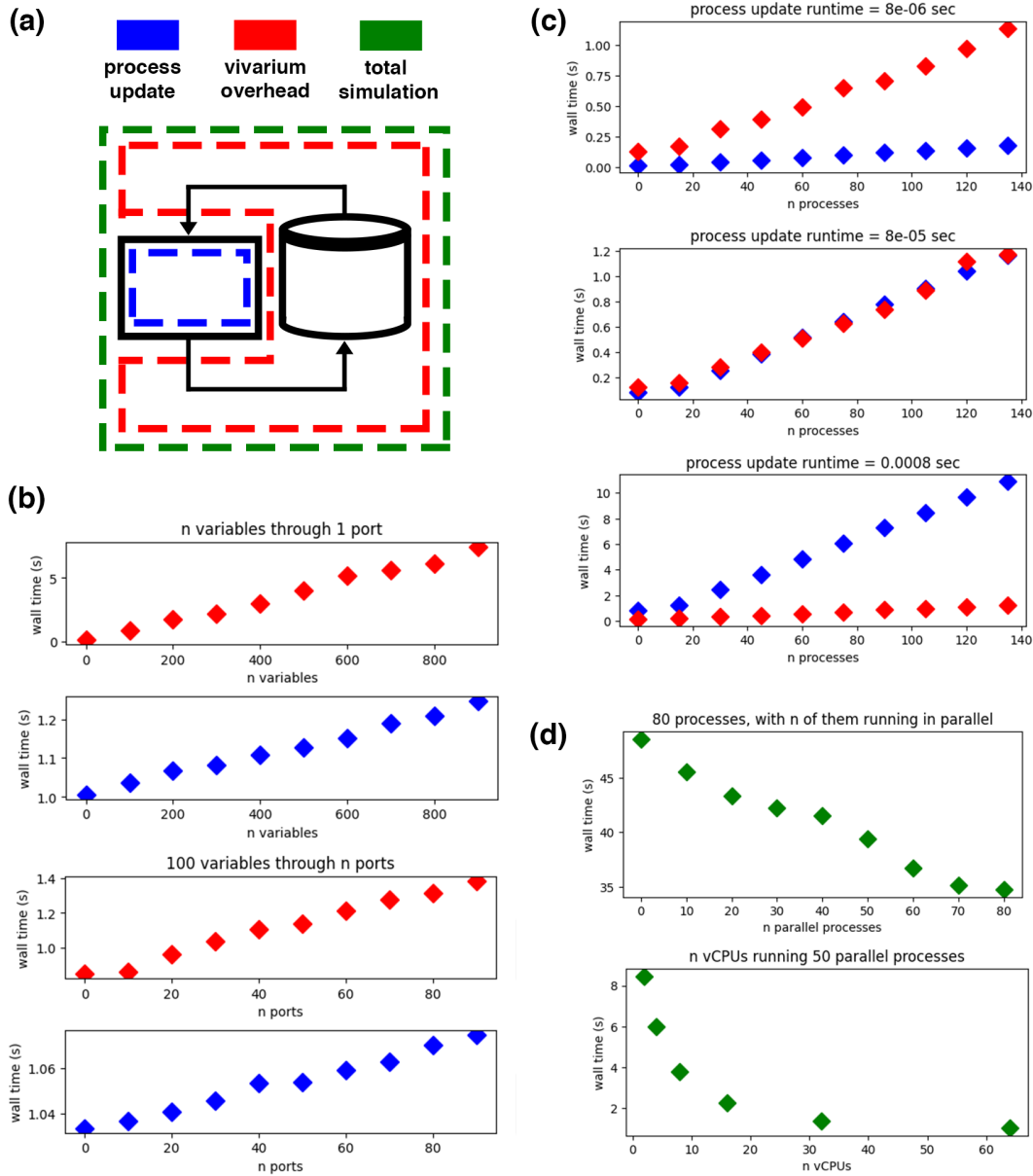
Figure S9: Performance evaluation **(a)** Schematic of computational costs: process update (blue) is the time spent by the processes, Vivarium overhead (red) is the time spent synchronizing the processes, total simulation time (green) could equal up to the combined process updates and Vivarium overhead. **(b)** Vivarium overhead scales linearly with the number of ports (top) and the number of variables per process (bottom) – process updates also scale linearly due to the extra time required to get random numbers and sorting them into ports. **(c)** Vivarium overhead and process update time scale linearly with the number of processes, but their relative scaling depends on the processes' computational requirements. The middle plot shows that the threshold of process update scalability is about 0.08 milliseconds ($8e-5$ seconds) on the computer used for this profiling run. **(d)** Simulation time decreases when processes are run in parallel. Top – 80 total processes are simulated with n of them made parallel, on a computer with a fixed number of CPUs. Bottom – 50 total parallel processes simulated on virtual machines with different numbers of CPUs.

5   Implementation details and support

This section provides details about additional coding methods for Vivarium, and technical specifications for the reported simulations.

5.1   Software availability

The Vivarium Collective is a hub for Vivarium modules, and includes all of the libraries used for this project. These libraries are listed in Table S7. All of the notebooks are available for Jupyter on the vivarium-notebooks repository. For easier accessibility, the work described in this manuscript is also available on Google Colab (Table S8).

The Vivarium Collective website: `https://vivarium-collective.github.io`.

Documentation for vivarium-core: `https://vivarium-core.readthedocs.io/en/latest/`.

Performance tests: `https://github.com/vivarium-collective/vivarium-core/blob/master/vivarium/experiments/profile_runtime.py`.

`https://github.com/vivarium-collective/vivarium-core/blob/master/vivarium/experiments/profile_image.py`.

Table S7: Vivarium libraries used in this project. These are all freely available for open development at the Vivarium Collective, and as pip-installable libraries on PyPI.

Library	Description
vivarium-core	The interface classes and multiscale simulation engine.
vivarium-notebooks	The repository developed for this paper, with Jupyter notebooks and Python files.
vivarium-bioscrape	For chemical reaction networks with SBML.
vivarium-cobra	For constraint-based models of metabolism, with BiGG model interface.
vivarium-multibody	Lattice composite model used for spatial multi-cell interactions.

Table S8: Python notebooks demonstrating paper simulations. Jupyter notebooks are available on GitHub, and require local installation. Colab notebooks run online, and no local installation is required.

Description	Link
Vivarium interface basics Jupyter notebook.	`https://github.com/vivarium-collective/` `vivarium-notebooks/blob/main/notebooks/Vivarium_` `interface_basics.ipynb`
Multi-paradigm composites Jupyter notebook.	`https://github.com/vivarium-collective/` `vivarium-notebooks/blob/main/notebooks/` `Multi-Paradigm-Composites.ipynb`
Building the lac operon models with BioCRNpyler Jupyter notebook.	`https://github.com/vivarium-collective/` `vivarium-notebooks/blob/main/notebooks/Lac_Operon_` `CRN.ipynb`
Vivarium interface basics Google Colab notebook.	`https://colab.research.google.com/drive/` `1-j2OXsflV8PZMbi9xldrCLkZ9Fj-y9sm?usp=sharing`
Multi-paradigm composites Colab notebook.	`https://colab.research.google.com/drive/` `1aiJ6uNjeATPOJTedxoZuNrLf3IPnHLU8?usp=sharing`

5.2 Tools for debugging

It is important to provide tools for debugging, so that users can more easily build models, find sources of error in the model, and feel confident in their results. Vivarium is a powerful tool to connect models together, but if the connection of processes' ports to stores is not done properly, the whole model would not work or generate spurious results. Therefore it provides some tools to aid in debugging the wiring of models.

There are two main tools at present. When initialized with processes and their topology, `Engine` checks whether the topology matches the ports structure. If all the ports are successfully maps, it generates the stores based on port schema specifications; if the ports are not all mapped, it quits and prints a message about which process and which of its ports were not successfully mapped by the user's topology declaration. This feedback can be used to ensure all ports are mapped to stores.

Another useful tool is a function called `plot_topology`, which is provided with vivarium-core (Fig S10). This function plots a composite's connectivity, with processes, stores and their connections. By seeing the connections visually, users can better track their model's structure and process interactions.
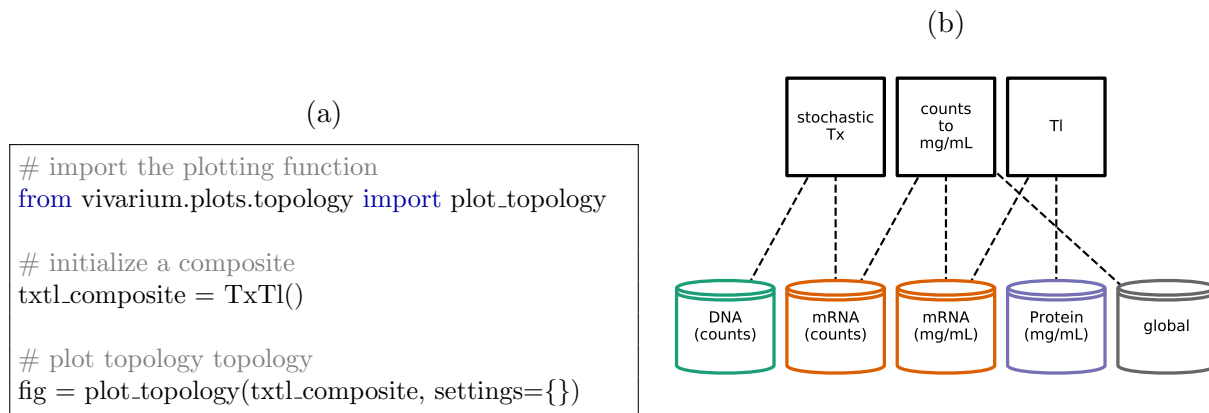
(b)

(a)

```
# import the plotting function
from vivarium.plots.topology import plot_topology

# initialize a composite
txtl_composite = TxTl()

# plot topology topology
fig = plot_topology(txtl_composite, settings={})
```

Figure S10: `plot_topology` is a useful tool for debugging by visualizing how processes and stores are connected. **(a)** Python script for generating a composite and plotting its topology **(b)** Example output, showing how three processes are connected to five stores.

5.3   Technical specifications.

The simulations shown in Section 3, and in supplemental Section 3 were run by executing the simulation functions in `bioscrape_cobra/simulate.py`, which can be found in the Vivarium-notebooks repository on GitHub. The simulations shown were run on a MacBook Pro, with a 3.1 GHz Quad-Core Intel Core i7 processor. The multi-paradigm composite simulation shown in Section 3 was completed overnight.

5.4   Emitters

There are different `Emitter` classes to save simulation output, which can be specified with arguments to `Engine`. `RAMEmitter` saves the simulation output to RAM – this is useful for small simulations that you might not need to return to. `DatabaseEmitter` saves simulation output to MongoDB. `NullEmitter` is an easy way to ignore all emits. Other emitters can be developed by subclassing the `Emitter` class.

5.5   Ports schema

Table S9 covers the different types of ports schemas, which are declared by the processes. Updater and divider schema have their own tables – S10 and S11 – with methods provided by vivarium-core. In addition to these provided methods, custom updaters and dividers can easily be defined by the user for any type of desired state.

26

Table S9: Schema. A process' `ports_schema` method declares the schemas for variables connected to each port. These are used to construct the stores, which apply the declared methods during runtime.

Attribute	Method
_default	The default value of the state variable if no initial value is provided. This also sets the data type of the variable, including units.
_updater	How to apply state variable updates. Available updaters are listed in Table S10.
_divider	How to divide the state variable's values between daughter cells. Available dividers are listed in Table S11.
_emit	A Boolean value that sets whether to log this variable to the simulation database for later analysis.
_properties	User-defined properties such as molecular weight. These can be used for calculating variables such as total system mass.

Table S10: Updaters available in vivarium-core. Updaters are methods by which an update from a process is applied to a variable's value. New updaters can be easily defined and passed into a port schema (see Listing S5).

Name	Function
accumulate	The default updater. Add the update value to the current value.
set	The update value becomes the new current value.
merge	Update an existing dictionary with new values, and add any newly declared keys.
null	Do not apply the update.
nonnegative_ accumulate	Add the update value to the current value, and set to 0 if the result is negative.
dict_value	translates _add and _delete -style updates to operate on a dictionary.

Table S11: Dividers available in vivarium-core. Dividers are methods by which a variable's value is divided when division is triggered. New dividers can be easily defined and passed into a port schema (see Listing S5).

Name	Function
set	The default divider. Daughters get the same value as the mother.
binomial	Sample the first daughter's value from a binomial distribution of the mother's value, and the second daughter gets the remainder.
split	Divide the mother's value in two. Odd integers will make one daughter receive 1 more than the other daughter.
split_dict	Splits a dictionary of *key* : *value* pairs, with each daughter receiving a dictionary with the same keys, but with each value split.
zero	Daughter values are both set to 0.
no_divide	Asserts that this value should not be divided.

5.6 Advanced coding methods

This supplementary subsection includes some code listings demonstrating more advanced methods for specifying models with Vivarium.

Listing S4: Declaring step dependencies. Steps are declared by the Composer method `generate_steps` – in the case below these are three steps called s1, s2, and s3. Steps can also be declared by `generate_processes` and will be pulled out by the Engine. The dependencies between steps are declared by `generate_flow` – each step name maps to a list of steps that it depends on. When the flow is triggered, the steps will run in an order declared by this dependency graph.

```python
def generate_steps(self, config):
    steps = {
        's1': Step(),
        's2': Step(),
        's3': Step(),
    }
    return steps


def generate_flow(self, config):
    flow = {
        's1': [],
        's2': [('s1',)],
        's3': [('s1',)],
    }
    return flow
```

Listing S5: Declaring custom updaters and dividers in `port_schema`. Updaters are functions that take a `current_value` and a `update_value`, and return the new value. Dividers are functions that take a `mother_value`, and a `state`, and return two values in a list – one for each daughter.

```python
# updater that returns a random value
def random_updater(current_value, update_value):
    return random.random()

# divider that returns a random value for each daughter
def random_divider(mother_value, state):
    return [
        random.random(),
        random.random()]

def port_schema(self):
    ports = {
        'port1': {
            'variable1': {
                '_default': 1.0
                '_updater': {
                    'updater': random_updater
                    }
                '_divider': {
                    'divider': random_divider
```

```
            }
        }
    }
    return ports
```

Listing S6: Using glob ('*') schema to declare expected sub-store structure. In this example, `port1` is connected to sub-stores specified by a glob schema. This allows the process to read anything that port1 connects to which adheres to its declared schema. Sub-stores can be added and removed during runtime, and the process will see it.

```
def port_schema(self):
    ports = {
        'port1': {
            '*': {
                '_default': 1.0
            }
        }
    }
    return ports
```

Listing S7: Declaring process-store connections with `generate_topology`. A topology is a Python dictionary with keys for processes, and subkeys for their ports which map to paths at which they will connect to stores. A flat network does not require a path, just a store name at the same level. The syntax used for declaring paths is a Unix-style tuple, with every element in the tuple going further down the path from the root compartment, and '..' moving up a level to an outer compartment.

```
def generate_topology(self, config):
    topology = {
        'process': {
            'port1': ('path','to','store'), # connect port1 to inner compartment
            'port2': ('..','outer_store') # connect port2 to outer compartment
        }
    }
    return topology
```

Listing S8: Splitting a port into multiple stores with `generate_topology`. Variables read through the same port can come from different stores. To do this, the port is mapped to a dictionary with a `_path` key that specifies the path to the default store. Variables that need to be read from different stores each get their own path in that same dictionary. This same approach can be used to remap variable names, so different processes can use the same variable but see it with different names.

```
def generate_topology(self, config):
    topology = {
        # split a port into multiple stores
        'process1': {
            'port': {
                '_path': ('path_to','default_store'),
                'rewired_variable': ('path_to','alternate_store')
            }
```

```
        }
        # mapping variable names in process to different name in store
        'process2': {
            'port': {
                '_path': ('path_to','default_store'),
                'variable_name': 'new_variable_name'
            }
        }
    }
    return topology
```

Listing S9: Store API. Stores can be easily constructed on-the fly using an experiment approach we call Store-API. This allows you to add processes and variables at different locations in a Store hierarchy, with easy path creation using indexing directly into a Store.

```
# create the root
store = Store({})

# create a new store at a path
store.create(['top', 'store1'])
store.create(['top', 'store2'])

# create a process at a path
store.create(['top', 'process1'], ToyProcess({}))
store.create(['top', 'process2'], ToyProcess({}))

# connect port using a relative path
store['top', 'process1'].connect('port1', 'store1')

# connect using store target through a different port
store['top', 'process1'].connect('port2', store['top', 'process1', 'port1'])

# connect using absolute path
store['top', 'process1'].connect('port2', ('top', 'store2'), absolute=True)

# put the store in the engine and run it
sim = Engine(store=store)
sim.update(100)
```

References

[1] Blomqvist, V. (2007–2019). Pymunk. http://www.pymunk.org/.

[2] Ebrahim, A., Lerman, J. A., Palsson, B. O., and Hyduke, D. R. (2013). COBRApy: constraints-based reconstruction and analysis for python. *BMC systems biology*, 7(1):74.

[3] Feist, A. M., Zielinski, D. C., Orth, J. D., Schellenberger, J., Herrgard, M. J., and Palsson, B. Ø. (2010). Model-driven evaluation of the production potential for growth-coupled products of escherichia coli. *Metabolic engineering*, 12(3):173–186.

[4] Gillespie, D. T. (2007). Stochastic simulation of chemical kinetics. *Annu. Rev. Phys. Chem.*, 58:35–55.

[5] King, Z. A., Lu, J., Dräger, A., Miller, P., Federowicz, S., Lerman, J. A., Ebrahim, A., Palsson, B. O., and Lewis, N. E. (2016). BiGG models: A platform for integrating, standardizing and sharing genome-scale models. *Nucleic acids research*, 44(D1):D515–D522.

[6] Orth, J. D., Thiele, I., and Palsson, B. Ø. (2010). What is flux balance analysis? *Nature biotechnology*, 28(3):245–248.

[7] pint (2021). pint. https://pint.readthedocs.io/en/stable/.

[8] Poole, W., Pandey, A., Tuza, Z., Shur, A., and Murray, R. M. (2020). BioCRN-pyler: Compiling chemical reaction networks from biomolecular parts in diverse contexts. *BioRxiv*.

[9] Santillán, M., Mackey, M., and Zeron, E. (2007). Origin of bistability in the lac operon. *Biophysical journal*, 92(11):3830–3842.

[10] Swaminathan, A., Poole, W., Hsiao, V., and Murray, R. M. (2017). Fast and flexible simulation and parameter estimation for synthetic biology using bioscrape.

[11] Varma, A. and Palsson, B. O. (1994). Stoichiometric flux balance models quantitatively predict growth and metabolic by-product secretion in wild-type Escherichia coli W3110. *Applied and environmental microbiology*, 60(10):3724–3731.

[12] Wong, P., Gladney, S., and Keasling, J. D. (1997). Mathematical model of the lac operon: inducer exclusion, catabolite repression, and diauxic growth on glucose and lactose. *Biotechnology progress*, 13(2):132–143.