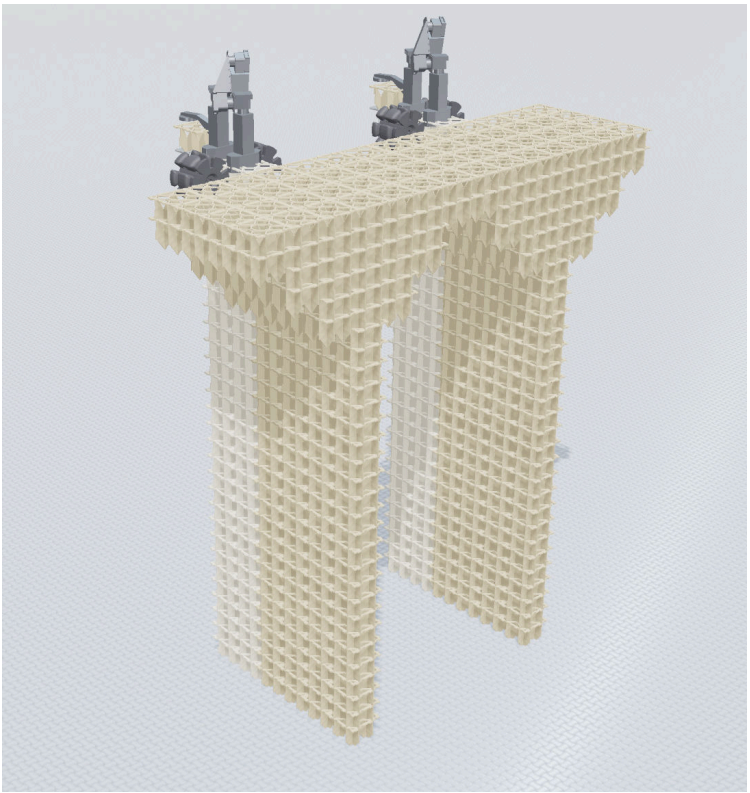
EPFL

THE CENTER FOR BITS AND ATOMS
Massachusetts Institute of Technology

MASTER THESIS

Feb. 2026 – Aug. 2026

A Physics-Aware Digital Twin for Robotic Lattice Assembly



Alessio Zazo (328450)

Robotics Master

Supervisors: Prof. Auke Jan Ijspeert, Prof. Neil Gershenfeld

August 13, 2026

Acknowledgments

This thesis was not supposed to happen this way. The plan was simple: fly to Boston, join the MIT Center for Bits and Atoms, and build robots. The embassy had other plans. After my visa was denied, this work turned into a thesis in three acts: three months in Rome, one month in Bhutan, and a final stretch in Lausanne, supervised across continents and through more Zoom calls than I can count.

Thank you Neil and Miana, for keeping me on the project despite the distance, for your guidance, and for treating a remote student like part of the team. Thank you Auke, for your support and supervision throughout this work, and for letting me settle into a desk at BioRob when I technically was not supposed to be there.

To the Bhutan team (Tenzin, Yeshy, John and Chirag), thank you for welcoming me, for everything you taught me on the ground, and for what was one of the most amazing experiences of my life.

Lausanne, August 13, 2026 Alessio Zazo (328450)

Abstract

Automated construction has mostly scaled by making the machine larger, so that a build is bounded by the envelope that can be transported to a site and erected on it. Distributed robotic assembly inverts that relationship. A fleet of small robots places discrete parts and climbs the structure it is assembling, so structure size is decoupled from machine size and throughput grows by adding robots rather than by enlarging one. Systems built on this principle have established that agents smaller than the artifact can assemble it, but their planners sequence placements geometrically. Where the parts carry no useful load the forces never arise, and where they do, what is verified is the finished structure rather than the states the build passes through. A partial structure is a load path under construction, and the robot standing on it is one of the loads that path has to carry. A sequence that is valid as geometry therefore passes routinely through states that would collapse, which is what holds distributed assembly at the scale of demonstrations rather than of load-bearing architecture.

What such systems lack is a planner for which the physics of every intermediate state is a constraint on what may be placed next rather than a check applied to a finished design. This thesis develops that planning layer as a physics-aware digital twin, faithful to the structure through a stability model built for the connection the parts actually use, and faithful to the robot through a validated kinematic model of the machine that places them. It is instantiated on a fleet of MILAbot inchworm robots assembling snap-fit timber voxels on a 75 mm lattice, although the formulation is not particular to that lattice. Its components are a stability model that returns a verdict per block and carries the weight of the robots, a definition of buildability that separates a structure standing when finished from one standing at every step, a reinforcement learning sequence planner whose action mask is the physics itself, a loop that grows an unbuildable target into a buildable one, and a pipeline running from a certified sequence to the physical robot.

On the benchmark suite, patterning a structure obtains stability an order of magnitude more cheaply than propping it, the learned policy leaves no unstable voxel where the other strategies fail, and the whole suite trains on a laptop without a GPU. All of it runs in one browser-based platform, in which a mesh is voxelised, its physics verified block by block, its sequence planned and played by a fleet, and sent to the physical machine from the same page. On hardware, that link drove the robot through simple cantilevered builds, placing the overhanging geometry the stability model exists to certify.

Contents

Chapter 1

Introduction

Machines built to automate construction scale by growing larger. A gantry spans only what it is built to span, an arm reaches only as far as its links allow, and every metre added to the envelope adds cost, transport, foundations and setup time on site. That relationship is the motivating constraint for this thesis, and it fits poorly with a programme such as Gelephu Mindfulness City in Bhutan, which spans more than a thousand square kilometres and is to be built in locally sourced wood, bamboo and stone [1, 2]. Building many medium-sized structures on a remote site from a local material supply is not work that suits one large machine.

Discrete, voxel-based robotic assembly inverts this relationship. The structure is decomposed into modular, reusable building blocks placed by a fleet of small mobile robots that climb what they have already built. Structure size is decoupled from machine size, throughput scales by adding robots rather than by enlarging one machine, and errors stay local, detectable and reversible because the material itself is discrete. Prior work at the MIT Center for Bits and Atoms (CBA) demonstrated this end to end, with MILAbot fleets assembling meter-scale lattice structures from an input mesh.

Deciding whether a fleet can build a given structure is not the same as deciding whether that structure will stand while it is being built. Distributed assembly systems, this one included, plan geometrically. They discretize a shape, generate a part pattern and sequence placements, with no notion of the forces acting on the partially built structure. Bridges, cantilevers, and overhangs routinely pass through intermediate states that would collapse under their own weight, or under the weight of the robot standing on them. A second failure mode is that the simulation drifts away from the hardware, so a plan that looks executable on screen is not executable by the machine that will actually build it. This thesis addresses both through a *physics-aware digital twin* that is faithful to the structure, through a stability model that evaluates every intermediate state, and faithful to the robot, through a validated kinematic model of the machine that places the parts.

1.1 Robotic Construction and Discrete Materials

Construction is among the least automated industries, and the attempts to change that split cleanly into two families. The first family scales the machine. Gantry and arm-based 3D concrete printing has reached commercial deployment along a well-charted research roadmap [3], and the Digital Construction Platform printed a 14.6 meter dome on site from a tracked vehicle [4]. Hadrian X lays conventional masonry from a truck-sized boom [5], and mobile robotic brickwork put industrial arms on unstructured sites, one robot to a structure [6]. These machines share the same ceiling, since the envelope bounds the build, deposition is irreversible, and an error is discovered downstream of the moment it was made (Figure 1.1).

Figure 1.1: Robotic construction systems in which the machine is larger than what it builds. **(a)** Hadrian X, a truck-mounted bricklaying robot [5]. **(b)** MaxiPrinter, a mobile 3D concrete printing robot [7]. **(c)** gantry-based 3D concrete printing of housing [8]. *Images: manufacturer and press material, as cited per panel.*

The second family sends agents smaller than the thing being built, and its reference point is biological rather than industrial. Termites raise mounds several metres tall with no drawing and nobody holding one, coordinating through the structure as it goes up [9]. TERMES made that principle mechanical, using a team of simple independent robots to assemble structures far larger than themselves [10]. Its foam blocks carry no load, so the structures remain demonstrations rather than load paths.

That last qualification separates the field into two halves that have not yet met. On one side sit the systems whose parts carry no useful load. TERMES stacks foam blocks [10], and aerial additive manufacturing prints a metre-scale cylinder in rapid-curing foam, and a smaller one in cementitious composite, from a team of flying robots [11]. Their planners are geometric by design, reasoning about reachability, traversability and deposition accuracy, since nothing in the demonstrated geometries makes force the binding question. The review of the field notes that structural performance is rarely what such demonstrations are built to test [9].

On the other side sit the systems whose parts do carry load. NASA's ARMADAS assembles

ultralight lattice structures whose finished stiffness and strength are the result being reported [12], and the hierarchical swarms of Abdel-Rahman and colleagues build from the same structural voxels their robots are made of [13]. What is analysed in that work is the completed structure. We are not aware of a system in either half whose *sequencer* is constrained by a force balance on every intermediate state. The previous MILAbot pipeline states the gap explicitly (Section 1.2), and the demonstrated geometries elsewhere are consistent with it.

The consequence is a ceiling on geometry rather than on hardware. A planner with no structural model can be trusted only on shapes that stand at every stage by construction, which in practice means wide bases, narrow tops, no unsupported span, and no state in which a robot stands on a cantilever. A planner that knows the forces lifts that restriction, since it can admit a span that is unsupported for twenty placements once it has certified that each of those twenty states holds, and refuse the single placement that would not. The same hardware then reaches bridges, cantilevered floors, deep eaves and enclosures rather than towers and walls. Nothing in that argument is specific to voxels, since it applies to any system that assembles discrete load-bearing parts with builders that climb their own work. A planner that reasons about the forces in the partial structure is the missing half of the case for distributed assembly.

What lets a system scale by adding builders and still carry load is a material designed to be assembled. *Digital materials* apply to structure the principle that bits apply to information, using a finite set of mass-produced part types joined at discrete relative positions [14]. Two properties follow from the discreteness itself. Accuracy is a property of the parts rather than of the machine placing them, so tolerance is set in the factory that makes the voxels and not on the build site. Additionally, a placement is binary, so an incorrect one is detectable and can be undone.

The blocks themselves, called *voxels* in this work, exist in many geometries and processes, from injection molded polymer frames to carbon fiber composites and folded metal sheet, spanning centimeter lattices to meter-scale assemblies (Figure 1.2a) [15, 16]. Because every part is identical and mass-produced, structures can be disassembled and rebuilt into new forms, and damage is repaired by exchanging cells rather than replacing the whole.

Cheung and Gershenfeld showed that such reversibly assembled cellular composites reach record stiffness-to-weight ratios, with bulk behavior predictable from part properties [14]. The same part logic provides function as well as structure. Indeed, the digital morphing wing replaces a monolithic airframe with a shape-adapting lattice [17, 18], and discrete mechanical metamaterials program rigid, compliant, auxetic and chiral behavior into the same parts [15].

Robots followed, and the material itself simplified their design. A periodic lattice is its own coordinate system, so a robot gripping it needs neither global localisation nor a structured factory floor. BILL-E, the Bipedal Isotropic Lattice Locomoting Explorer, set the canonical form, an inchworm that walks on the lattice it builds [19, 20]. The idea has since travelled well beyond the CBA, to the inchworm pairs of NASA's ARMADAS, which check for error at every step [12], and to the hierarchical

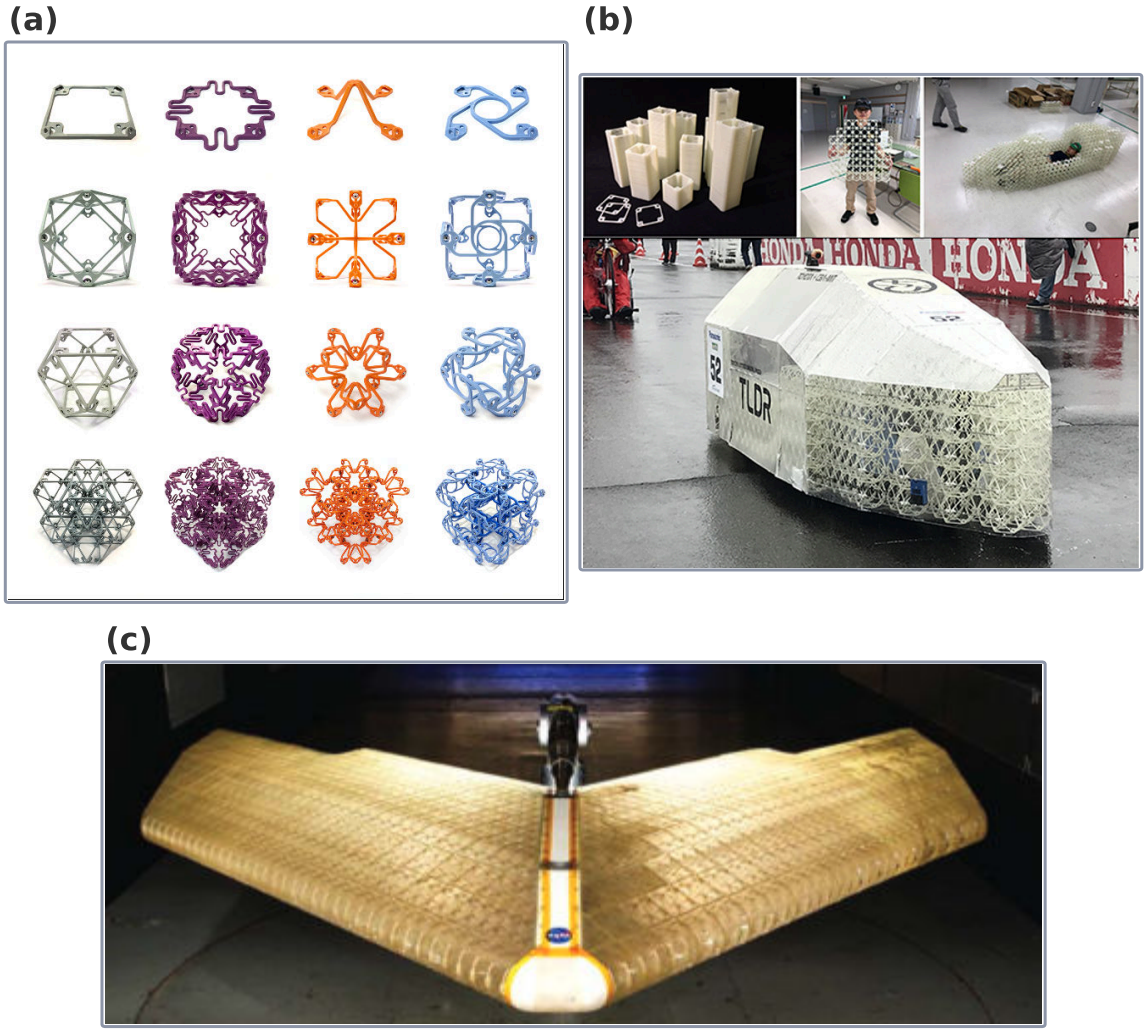


Figure 1.2: Voxels as a material system. **(a)** an injection-molded voxel family whose part types span rigid, compliant, auxetic and chiral behavior [15]. **(b)** 3D printed voxels assembled into a super-mileage vehicle body, where the same cells disassemble and rebuild into other machines *(image: MIT CBA)*. **(c)** the digital morphing wing, a shape-adapting lattice aircraft structure [17]. *Images as cited per panel.*

self-replicating swarms of Amira Abdel-Rahman and colleagues [13].

1.2 Prior Work on MILAbot

The system this thesis extends was built at the CBA in two halves, one physical and one computational.

The robot and the material. Miana Smith built both, and built them to fit each other, from the first MILAbot generation through the v2 used here, together with the voxel set and the elevator. Robots are assembled from the same voxels they place, joined reversibly and strongly enough that one robot can build another [21]. The material system is hierarchical, with parts small enough for ordinary digital fabrication compounded into blocks that mobile robots then assemble [22]. The MILAbot itself is an inchworm with grippers at both ends and voxel carriers on its back (Figure 1.3a).

Two findings from that work constrain any planner written on top of it. The lattice's alignment features absorb placement error of roughly half a pitch, which is what lets a low-cost machine build accurately without metrology. Additionally, a robot walking repeatedly over the lattice fails about 7% of the time [21]. Taken together, these findings mean that the robot is a load on its own work and that moving it around is not free.

The pipeline. Paul Richard's thesis is the software counterpart and the direct foundation of this one, providing an end-to-end path from mesh to machine that covers voxelisation, block-pattern and support generation, task allocation across a fleet, path planning and multi-robot hardware assembly (Figure 1.3b) [16]. It ran on meter-scale builds, and it closed with the comparison that remains the clearest argument for the whole approach, showing that a handful of MILAbots match arm-based concrete printing at roughly one hundredth of the cost.

That pipeline reasons about geometry. Stability is inferred from connectivity, and its author names the limit himself, writing that "this algorithm doesn't have any stability check as we assume that inputted structures are stable". He lists a structure-aware algorithm "ensuring the stability of the structure during and after the assembly" as future work [16], and that omission is the first of the two gaps this thesis addresses.

The second gap is fidelity, and the same thesis records why it matters. Only one of its hardware demonstrations completed without failure, while the rest lost gears, deformed under load, developed backlash, or lost a motor zero [16]. Every one of those failures invalidates a hard-coded joint position, which is precisely what a plan written against an idealised kinematic model assumes will hold. The hardware has since moved on a full generation (Chapter 2), so the previous simulator no longer describes the machine that would execute its plans.

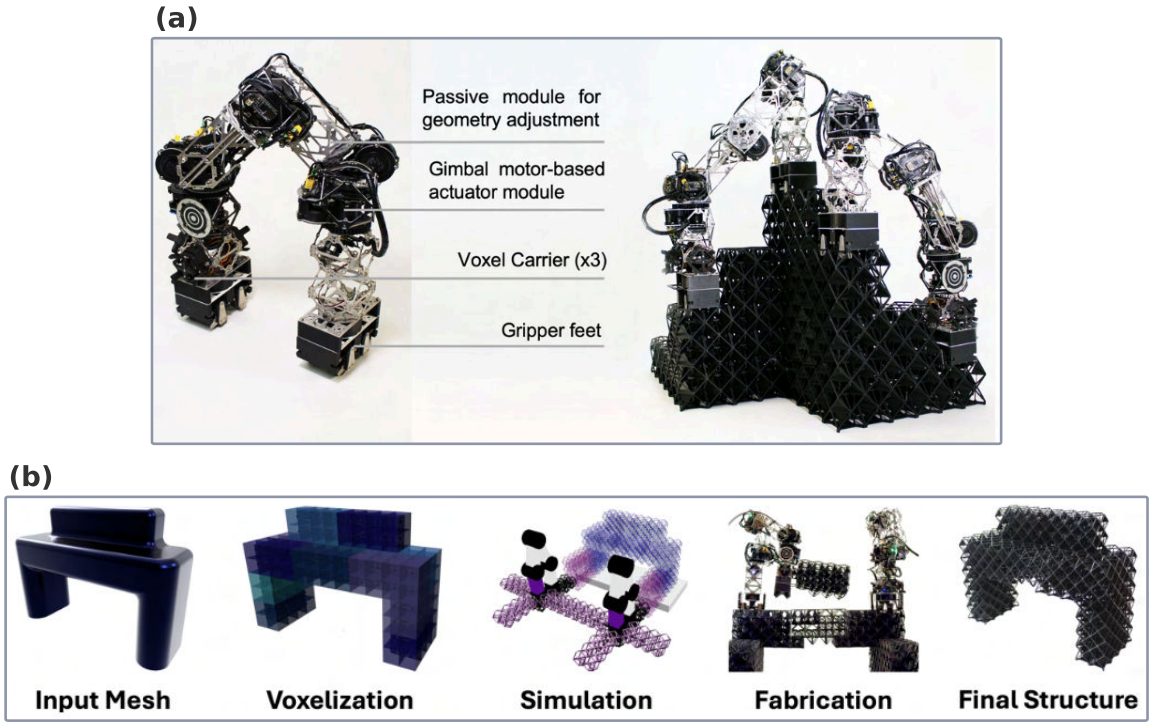


Figure 1.3: The MILAbot system this thesis extends. **(a)** the robot and its four module types, actuated joints, passive spacers, voxel carriers and gripper feet (left), and two MILAbots climbing a voxel structure (right). **(b)** the previous geometry-driven pipeline, from input mesh through voxelization and simulation to fabrication and the final structure. *Images: [16].*

1.3 Gelephu Mindfulness City

Gelephu Mindfulness City is a special administrative region in southern Bhutan, masterplanned by Bjarke Ingels Group with Arup and Cistri [1, 23]. It is a construction programme rather than a building. It comprises eleven neighbourhoods of repeating typologies around central public spaces, civic functions carried on inhabitable bridges over the rivers crossing the site, and a material ambition stated as explicitly as the spatial one, in the vernacular grammar of *rabsel*, cornices and tiered roofs [1, 2].

The hardware is aimed at this application. In 2025 Druk Holding and Investments and the MIT Center for Bits and Atoms signed an agreement to develop voxel-based robotic construction for the programme, run through the Jigme Namgyel Wangchuck Super Fab Lab [24]. Read as an engineering brief (Figure 1.4), the programme has three properties that matter here.

First, it is many similar structures rather than one landmark. That is the regime in which the

Figure 1.4: Gelephu Mindfulness City as an engineering brief. **(a)** one neighbourhood of eleven, with repeating typologies around a single centre. **(b)** civic buildings under lattice facades and deep tiered roofs. **(c)** an inhabitable bridge carrying civic programme over water. **(d)** a street under a full-depth eave. **(e)** the airport, a stacked timber roofscape. *Images: [23], masterplan by BIG with Arup and Cistri, reproduced here as a design brief.*

patch decomposition, symmetry detection and warm-started policies of Section 3.3.4 are worth their cost, since a solved patch is reused rather than re-solved, and in which fleet throughput rather than single-machine speed sets the schedule (Section 5.1).

Second, a snap-fit timber lattice suits the stated material ambition, since it is assembled rather than cast and reversibility is a property of the material system rather than an end-of-life plan.

Third, the vernacular is built from stacked cantilevers, namely lattice facades and deep tiered roofs, an eave carried the full depth of a street, a bridge held clear of the water, and an airport that is mostly roof. These are the geometries on which a geometry-driven sequencer dead-ends mid-build, which is what the stability model of Section 3.1 and the target evolution of Section 3.4 exist to handle.

1.4 Contributions

Figure 1.5: The stability-aware pipeline of this thesis, end to end. **(1)** the input mesh, the target geometry. **(2)** sequence planning decides in what order blocks are placed. **(3)** the stability model certifies that every intermediate state stands. **(4)** robot planning resolves stances, paths and the fleet assignment, generating climb supports as it goes. **(5)** the digital twin executes the plan, checking that every stance and swing is reachable and collision-free. **(6)** the same plan drives the physical robots.

Figure 1.5 shows the planning layer this thesis develops, applied to the MILAbot system. All of

it runs in one browser-based platform, where a structure is voxelised, its physics verified, and its build sequence planned and played, and from the same page the plan drives the physical robot. The contributions are:

- **C1. Stability model.** For any assembly state, a force balance determines whether the structure holds and names the block and the connection that fail if it does not. The connections are capacity-limited, so the verdict is per block rather than one global number a planner cannot act on (Section 3.1).
- **C2. Sequence planning.** A build order is produced in which every intermediate state is stable and every placement is one the robot can reach, using a learned policy whose action mask is the physics itself. Patch decomposition and macro placements carry it from toy grids to architectural scale (Section 3.3).
- **C3. Target evolution.** When a geometry admits no stable build order at all, a detect, grow and rebuild loop adds material and never removes any, until the unmodified planner covers the original shape completely (Section 3.4).
- **C4. Digital twin of MILAbot v2.** A simulator was rebuilt to reproduce how the current machine walks, reaches and places, with kinematics validated against the hardware, a locomotion catalog checked frame by frame, and placement gestures forward-simulated for collisions before they run (Chapter 4).
- **C5. Village-scale pipeline.** The pipeline provides a connected path from analysis to simulation, with no manual re-entry between the two, fleet coordination with per-structure robot teams, and many structures built in one scene, with the cost of real-robot constraints measured rather than assumed (Section 5.1).
- **C6. Hardware execution.** A live bridge drives the physical robot as the build plays, naming motions the robot already owns instead of streaming trajectories to it (Section 5.3).

The central claim of this thesis is that voxel-based robotic construction becomes reliable when planning is grounded in a digital twin that is faithful to both the physics of the structure and the capabilities of the robot.

Chapter 2

System Overview

Both problems named in Chapter 1 were hardware problems before they were software ones. The previous machine lost printed gears, deformed under load and drifted off its motor zeros, so the joint positions its plans were written against stopped being true [16]. The machine and the voxel have both been redesigned since. The v2 runs closed-loop on bought harmonic drives, and the block is now a timber part sized for the application of Section 1.3.

The construction voxel, the MILAbot v2 robot and the elevator are not the work of this thesis, since the hardware is Miana Smith's [25]. The planner is written against their constraints, so those constraints have to be stated before any of the planning makes sense.

2.1 Construction Voxel

The building block is a discrete voxel on a 75 mm pitch, assembled from flat sheet stock rather than moulded or printed as a volume (Figure 2.1a, b). A block is made from three part types, a width part, a length part and a top part, repeated as many times as the block's dimensions require. Every part is installed from the same direction, with a single final part inserted orthogonally to constrain the assembly. That low degree-of-freedom sequence is what makes the block itself robotically assemblable, while the flat feedstock is what makes it transportable, so that a wall arrives as sheets and becomes volume on site.

Three features do the structural work. Pyramidal faces on the top surface act as alignment features into the block above and carry shear. A snap fit at the base of each pyramid finally constrains the block's position, and it is that capacity, $T_{\text{snap}}^{\max}$, that Section 3.1 takes as the governing parameter of the stability model. Dovetail features along the upper face transmit lateral loads between neighbours.

Individual unit cells are grouped into larger blocks, because a robot placing a four-cell block

instead of four cells does a quarter of the work. The catalog the planner draws on holds sixteen, listed in Table 2.1: three single-course footprints (2×2, 2×3, 2×4), the same three at double height, and ten pre-stacked staggered blocks whose upper course is offset by one or two cells in x, in y, or in both. Tiling blocks so that they overlap offsets the seams of their joints, which is the mechanism behind the staggered bond patterns of Section 3.3, where the same overlap that helps the robot also improves load transfer.

Footprint	Single course	Double height	Staggered
2×2	`2x2`	`2x2x2`	`2x2x2 o1`, `2x2x2 o1-xy`
2×3	`2x3`	`2x3x2`	`2x3x2 o1-x`, `2x3x2 o1-y`, `2x3x2 o2-y`, `2x3x2 o1-xy`
2×4	`2x4`	`2x4x2`	`2x4x2 o1-x`, `2x4x2 o1-y`, `2x4x2 o2-y`, `2x4x2 o1-xy`

Table 2.1: The sixteen catalog blocks. A name gives the footprint in cells and, where a third figure is present, the number of courses, so that a $2 \times 4 \times 2$ block covers eight cells per course over two courses. The suffix `o1` or `o2` gives the offset of the upper course in cells and the trailing letters give the axis or axes it is applied along. The 2×2 footprint is symmetric, so an offset in x and an offset in y give the same block and the axis is left off the name.

The simulator carries each of those blocks in two representations. The STL view renders the real part geometry and is what the robot's gestures and the stability analysis are checked against. It is also too heavy to draw a building out of, so above a few hundred blocks the scene falls back to a simplified view in which each block is a solid volume on the lattice. Nothing about the plan changes between the two representations, only what is drawn.

2.1.1 From Mesh to Cells

The first stage of the pipeline turns an input geometry into a set of occupied cells on the lattice. The input is a 3D mesh in STL format, and the voxeliser computes the axis-aligned bounding box of the mesh, lays a uniform grid over it at the 75 mm pitch, and decides for each grid point whether it falls inside the mesh by casting rays along the coordinate axes and taking the majority verdict. The output is a list of occupied cell centres.

That stage is inherited rather than developed here. It comes from the hierarchical discrete lattice assembly work of Smith, Richard, Kyaw and Gershenfeld [22], with the algorithm as implemented by Richard in the previous pipeline [16].

Deciding which cells to occupy and deciding what to place on them are separate problems. The previous pipeline answered the second by covering the cells with catalog blocks layer by layer, and it is that answer, not the voxeliser, that Section 3.2 and Section 3.3 replace.

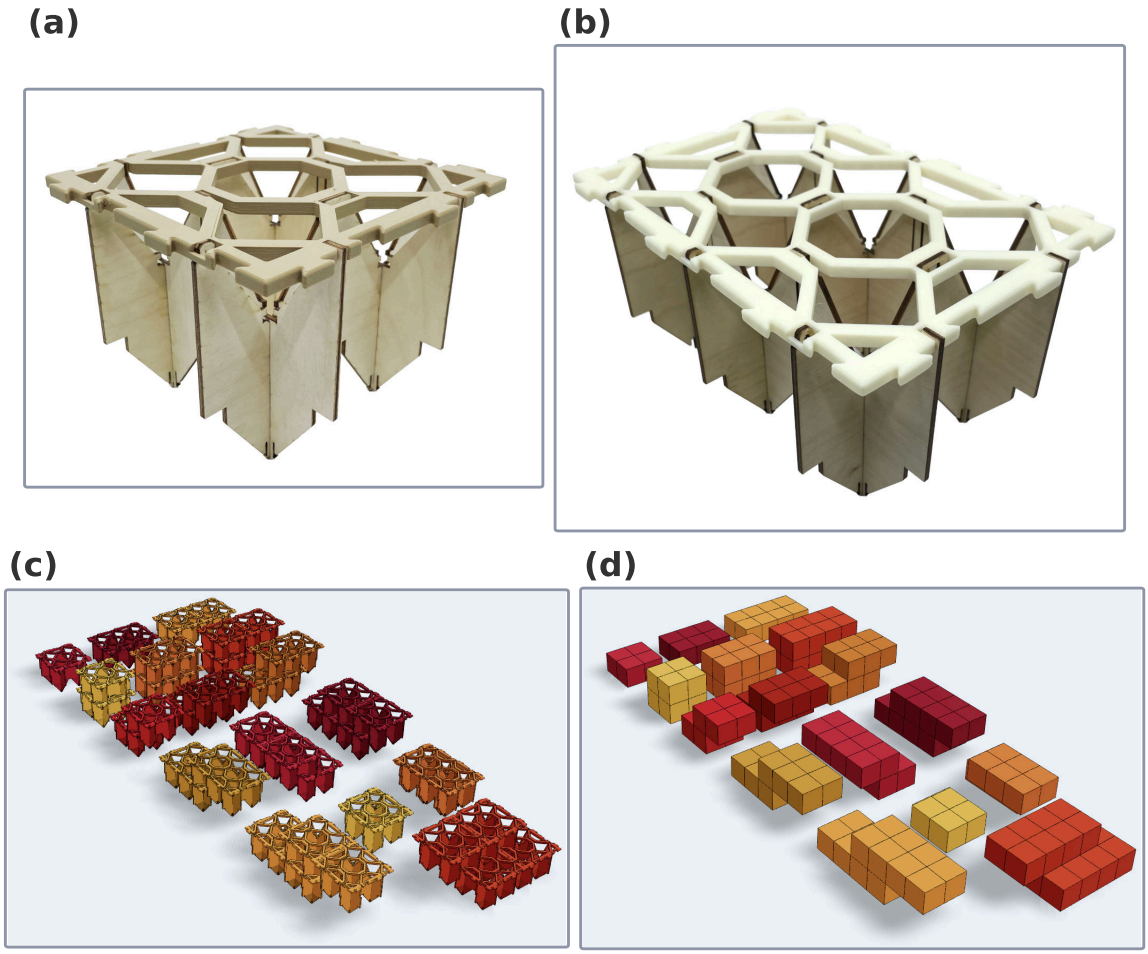


Figure 2.1: The construction voxel, as built and as simulated. **(a)** a 2 × 2 block, all parts cut from plywood. **(b)** a 2 × 3 block whose top part is printed. **(c)** the sixteen-block catalog in the simulator's STL view. **(d)** the same catalog in the simplified view. Colour distinguishes blocks, nothing more. Hardware and photographs: Miana Smith [25], with renders from this thesis's simulator.

2.2 MILAbot v2

MILAbot is a relative assembler, walking on the structure it is building and taking its global accuracy from its local relationship to the lattice underneath it. A gantry's precision is bounded by the size of the gantry, so that building something larger means building a larger machine, while a relative assembler inherits the accuracy of a structure that is mechanically error-correcting by construction. That relationship is also why the planner in this thesis has to reason about where the robot stands, since the machine is part of the load path rather than an observer of it.

The v2 is a two-gripper inchworm built around a custom actuator module (Figure 2.2c). Each joint is a BLDC gimbal motor driven through a 100:1 harmonic reduction, closed-loop on a magnetic encoder, with the controller board packaged inside the same voxel-sized envelope. The detail that matters for packaging is the feedback path, where the output shaft is run backwards through the motor's own hollow shaft, so the motor and output encoders sit on one axis instead of being offset as they were in v1. That offset introduced collisions and constrained the reachable poses. Modules take target positions over I2C and report encoder readings back. The robot as a whole receives its commands over WiFi, which is the link Section 5.3.2 describes.

The locomotion feet latch onto the lattice with aluminium "toes" driven by a servo through a rack and pinion, extending under a voxel beam to engage and retracting to release. The base of each foot carries pyramidal features that index into the voxel top surface, and those alignment cones absorb roughly ±25 mm of placement error. The v2 foot locomotes over a two-voxel-wide profile, where v1 required a 2×2 footprint for every step. A separate payload gripper latches into the central void of a 2×2 block and is offset behind the foot, so a carried block can be presented at several orientations.

In the twin the same machine is a five-joint kinematic chain with the two feet and the payload gripper modelled as the volumes they occupy, since reach and self-collision decide whether a planned placement is executable. What the robot can do on the lattice is a finite list, because the lattice is finite and a foot either lands in the middle of a 2×2 cell group or does not land at all. Table 2.2 summarises the vocabulary, while Chapter 4 develops the twin model and gives the catalog family by family with the validation behind it.

Movement	**Description**
Step forward ($z = 0, \pm\frac{1}{2}, \pm 1$)	Walks one voxel along the lattice, on the level or climbing onto and off a step half a voxel or a full voxel high.
Side-step ($z = 0, -\frac{1}{2}, -1$)	Travels laterally, left or right, one voxel or two, level or descending.
Turn left / right ($z = 0, \pm\frac{1}{2}, \pm 1$)	Pivots 90° about the planted foot, on the level or while gaining or losing height.
Crest	Crosses a ridge or a dip in one motion, rather than a climb up followed by a climb down.
Switch legs	Swaps which gripper leads, so every other movement is available in both leg parities.
Pick up / hand off	Takes a block from a stock stripe with the payload gripper, or receives one from another robot.
Place ($z = -1$ to $+2$)	Sets the carried block into the structure, using one of twenty-one placement gestures chosen by which approach the surrounding terrain leaves open.

Table 2.2: The MILAbot v2 movement vocabulary. Heights are relative to the stance the robot starts from, in voxels. Table 4.1 breaks the locomotion half of this list into its seven families and counts the actions in each.

The robot weighs about 4.5 kg and carries one block at a time, and a movement, a full step forward or a placement, takes on the order of ten seconds. That second number is what turns a placement count into a build time in Chapter 6.

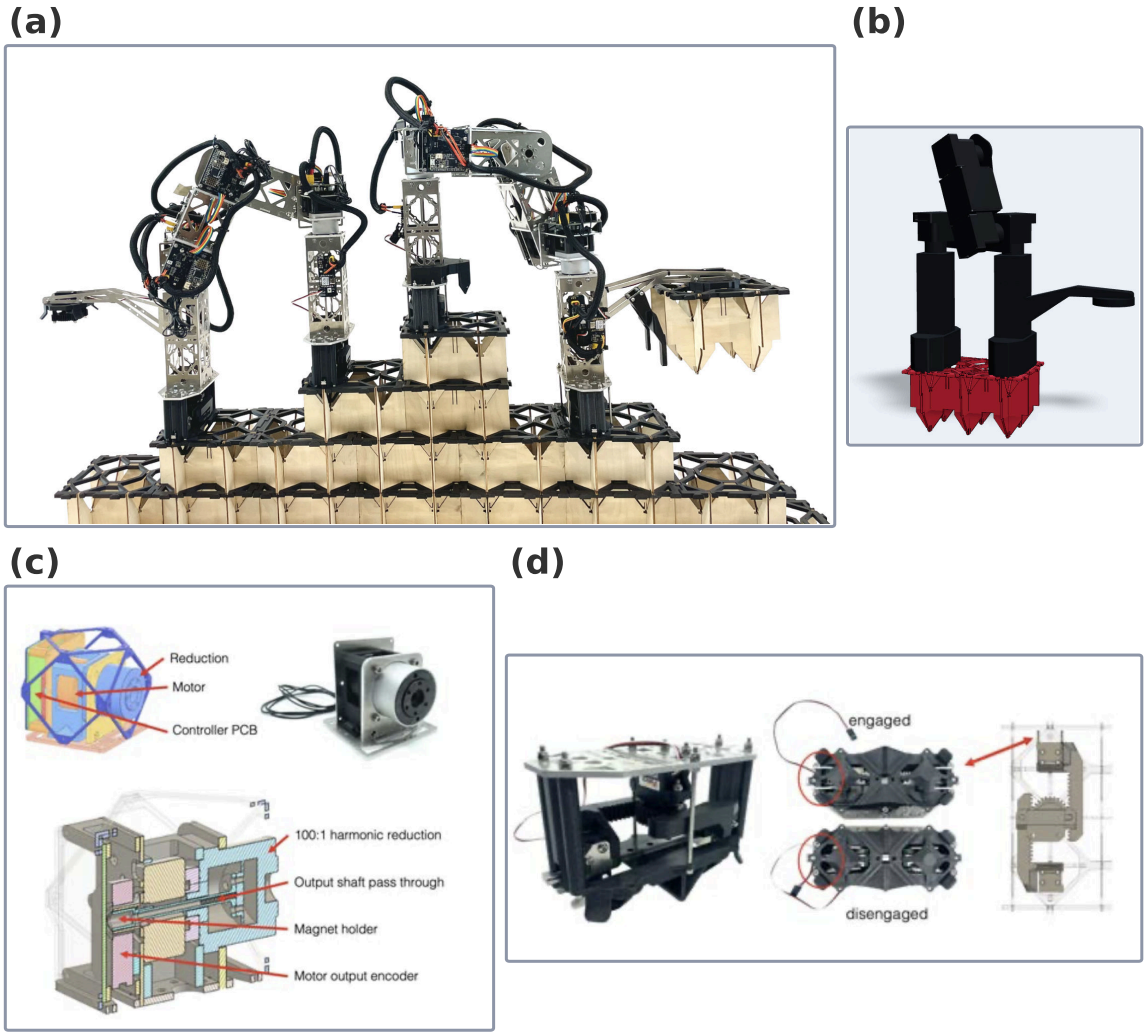


Figure 2.2: MILAbot v2, as built and as simulated. **(a)** two robots working on a voxel structure, the configuration everything in this thesis plans for. **(b)** the same machine in the digital twin, standing on a block. **(c)** the custom actuator module, with the cutaway showing the output shaft passing back through the motor. **(d)** the locomotion gripper, engaged and disengaged. Hardware, CAD and photographs: Miana Smith [25], with the render from this thesis's twin.

One more machine belongs in this picture, because the blocks the MILAbot places do not arrive assembled either. Figure 2.3 shows the cell that turns flat parts into blocks, where side faces and top faces are presented in separate feeds, a jig holds a block square while it is built up, and a pincer

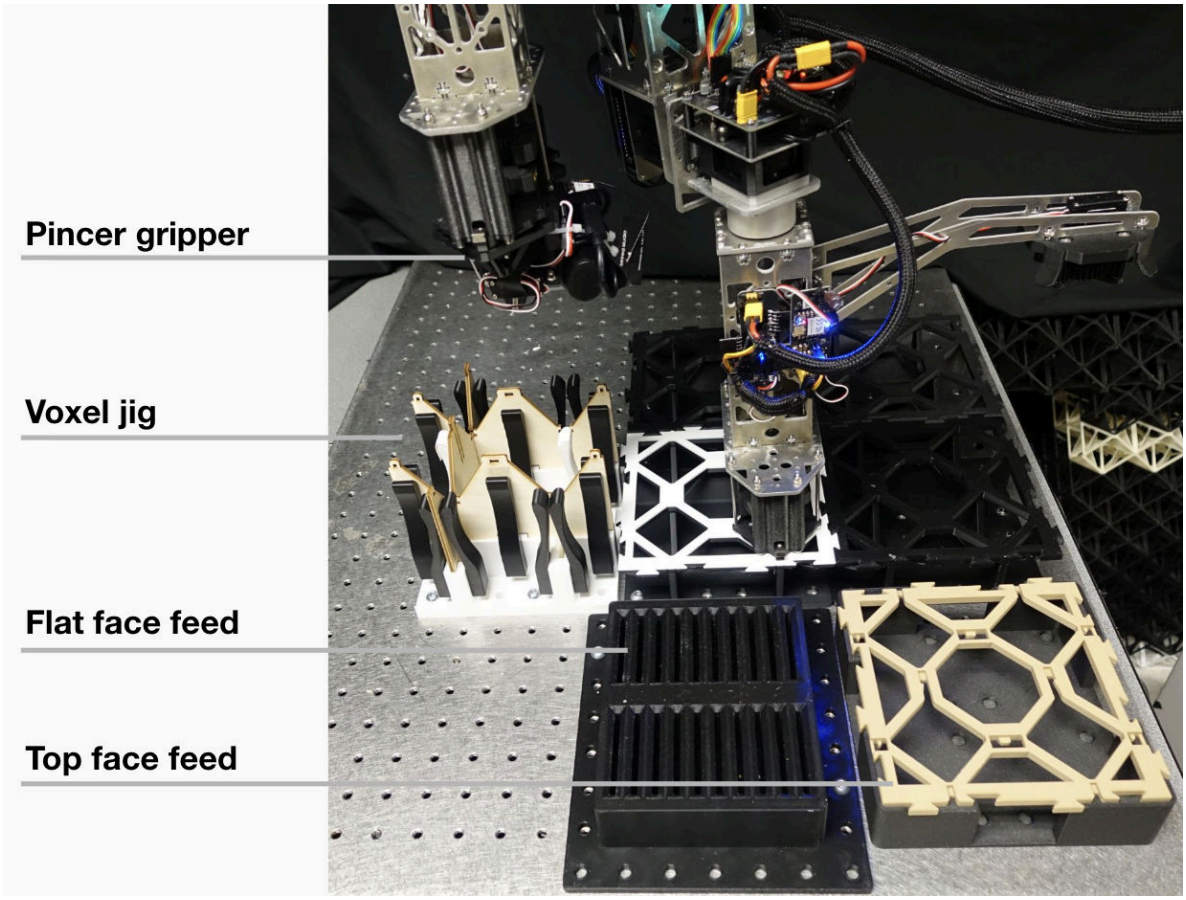


Figure 2.3: The voxel assembly cell, in which blocks are built from flat parts by a robot rather than by hand. Hardware and photograph: Miana Smith [25].

gripper installs each part. The single-direction discipline of Section 2.1 therefore runs at both scales.

2.3 Elevator

The blocks start on the ground, and carrying every one up by inchworm would dominate the build. The elevator addresses this, straddling a dedicated climb column and ferrying voxels from ground stock to the working course (Figure 2.4). Two sets of lobed drive wheels engage the column faces, so the machine climbs the same lattice the robots build with rather than needing a track of its own.

The planner has to account for the climb. The climb column is not scaffolding put up and taken down at will, but a structure that must exist before the course above it can be reached, and it rises with the build. Section 5.1.2 develops the climb-support system that keeps it connected to the

structure at every stage.

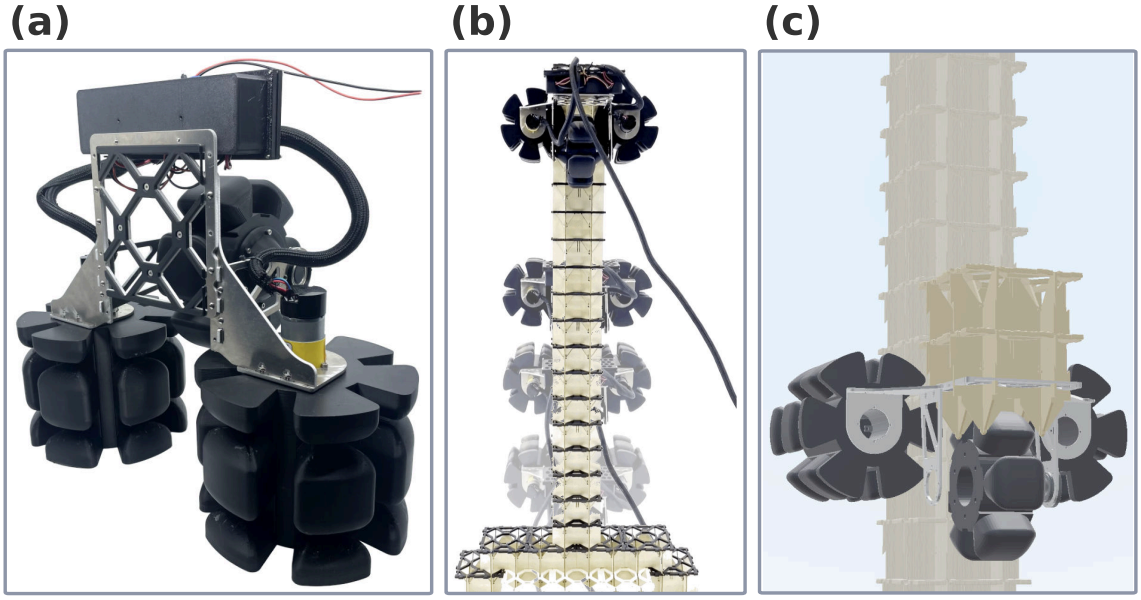


Figure 2.4: The elevator. **(a)** the lobed drive wheels. **(b)** one climb of the real machine, four frames of the same ascent composited into one image, with the earlier positions ghosted back. **(c)** the same machine straddling its climb column in the simulator. Photographs: Miana Smith [25].

Chapter 3

Stability-Aware Planning

Whether a structure stands has to be decided at every step of its own construction, not only once it is finished, and under the weight of the robots doing the building. This chapter develops that check, compares the two remedies available when it fails, and then turns the check itself into a constraint on the build order.

3.1 Stability Model

Before a build order can be generated, a definition of stability is needed, as every downstream decision depends on whether the partial structure stands. A voxel assembly is a set of separate blocks held to one another by snap fits, so a useful definition has to name which block is not held, and by how much the joint holding it is over capacity, rather than return a single number for the whole structure.

A build order is valid only if every state along it stands, and a robot may step only where the blocks will hold it, so that definition is evaluated thousands of times per run. Such a definition requires committing to how one voxel holds another. The MILAbot block stacks through an interlock at the center of its top face and is retained by snap-fit clips at the edges, and it is those clips that give way first. The model below adapts StableLego [26] to that contact geometry and extends it with the weight of the robots doing the building. It is a single program, implemented in `pipeline/backend/stablelego`, and every verdict reported in this thesis comes from it.

3.1.1 Why Not Finite Elements

Finite-element analysis meshes the voxels as a solid, fixes the grounded cells, applies self-weight and robot loads, and solves. We implemented it following the reduced-order beam formulation Amira Abdel-Rahman developed to give designers fast structural feedback on voxel assemblies [27], and the analyzer keeps it as a cross-check. It is not the planner's oracle, however, for two reasons.

The first is that FE checks a criterion this system never reaches. Treated as a continuum, a column of lattice fails in compression when $\rho g H \geq C_{\max}/a^2$, whereas a vertical interface instead parts in tension once the suspended weight exceeds the three snap fits holding it, $3T$. With the constants of Appendix A,

$$H^*_\sigma = \frac{C_{\max}}{\rho g\, a^2} = 1324 \text{ m}, \qquad H^*_{\text{joint}} = \frac{3T}{\rho g\, a^2} = 3.18 \text{ m}, \tag{3.1}$$

a factor $C_{\max}/3T = 417$ apart. Every structure in this thesis lives near the second limit and nowhere near the first. Lowering the allowable stress does not close the gap, as linear FE ties the interfaces into one bonded body and therefore transmits tension across a joint that has already parted. Its error is consequently one-sided, missing an instability but never inventing one, which is the wrong direction for an oracle whose job is to reject sequences that collapse.

The second is what each model returns (Figure 3.1). FE gives fields over a mesh, which a planner can only reduce to a single global number, and the available candidates are ill-posed. Indeed, elastic displacement converges to a nonzero value, so any cutoff is arbitrary, and peak von Mises stress does not converge at all at the re-entrant corners of a lattice. The force balance instead answers per block and per connection. That per-block answer is what the sequence planner's mask and the target-evolution loop consume (Sections 3.3 and 3.4), and it is mesh-independent by construction.

3.1.2 Related Work: Stability of Block Assemblies

In computer graphics, Whiting et al. formulated masonry stability as the existence of a set of compressive interface forces in static equilibrium, solved per structure as an optimization [28], the conceptual template for the force-balance methods that followed, and Make It Stand carried the same standing-under-gravity criterion into fabrication-aware design by deforming shapes until they balance [29]. Closest in spirit to construction planning, Deuss et al. computed assembly sequences for self-supporting masonry structures in which every intermediate state is stable, inserting temporary supports exactly where the partial structure cannot hold itself [30], which prefigures the support blocks of Section 3.2.

For block assemblies specifically, StableLego is the closest prior work, a per-brick force-balance model for LEGO with pressing, pulling, supporting and dragging forces at knob contacts (Figure 3.3) [26]. Its contact model is LEGO-specific in two ways that matter here, as the knob contacts

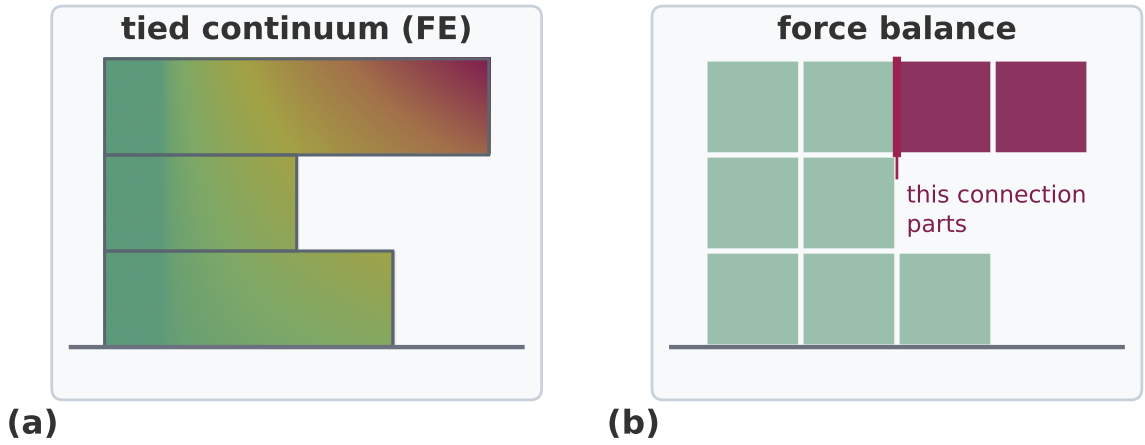


Figure 3.1: Output of each model on the same overhanging structure (schematic). **(a)** a tied-interface continuum gives one field over one bonded body. **(b)** the rigid-body force balance gives a verdict per block and a force margin per connection, showing which two blocks cannot be held and which interface parts.

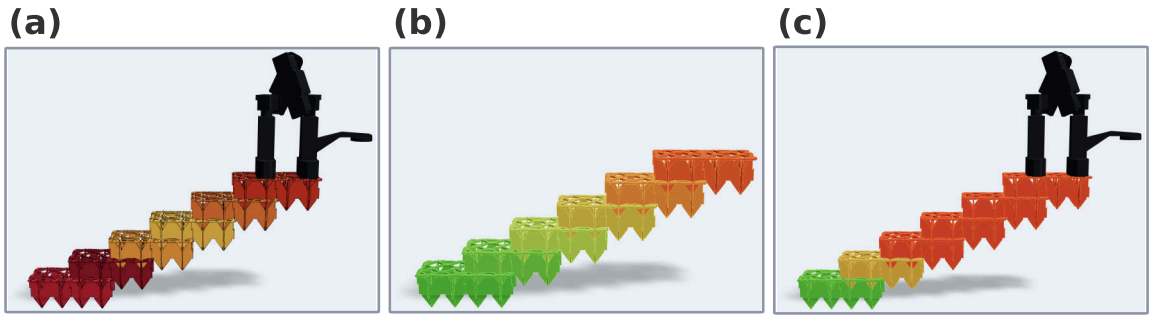


Figure 3.2: The analyzer's FE cross-check on the staircase benchmark, after Abdel-Rahman's formulation [27] (tied-interface continuum, $E = 0.2$ GPa, $\nu = 0.36$, $\rho = 137\,\mathrm{kg\,m^{-3}}$). **(a)** the loaded state, blocks colored by identity. **(b)** the displacement field on the bare structure. **(c)** the same field with the robot on the top step, sharing one fixed scale with (b), green to red over 0 to 0.0413 cm. Even loaded, the deflection stays a fraction of a millimeter per joint, yet the force balance calls that same state four-of-six blocks unstable.

of a connection spread across the whole footprint of the brick, and the tension a connection carries is left unconstrained in the force solve, with the measured knob capacity checked only afterwards in the stability criterion. Neither assumption holds for the MILAbot voxel, and here the structure additionally has to carry the machines assembling it. Table 3.1 sets the two models side by side.

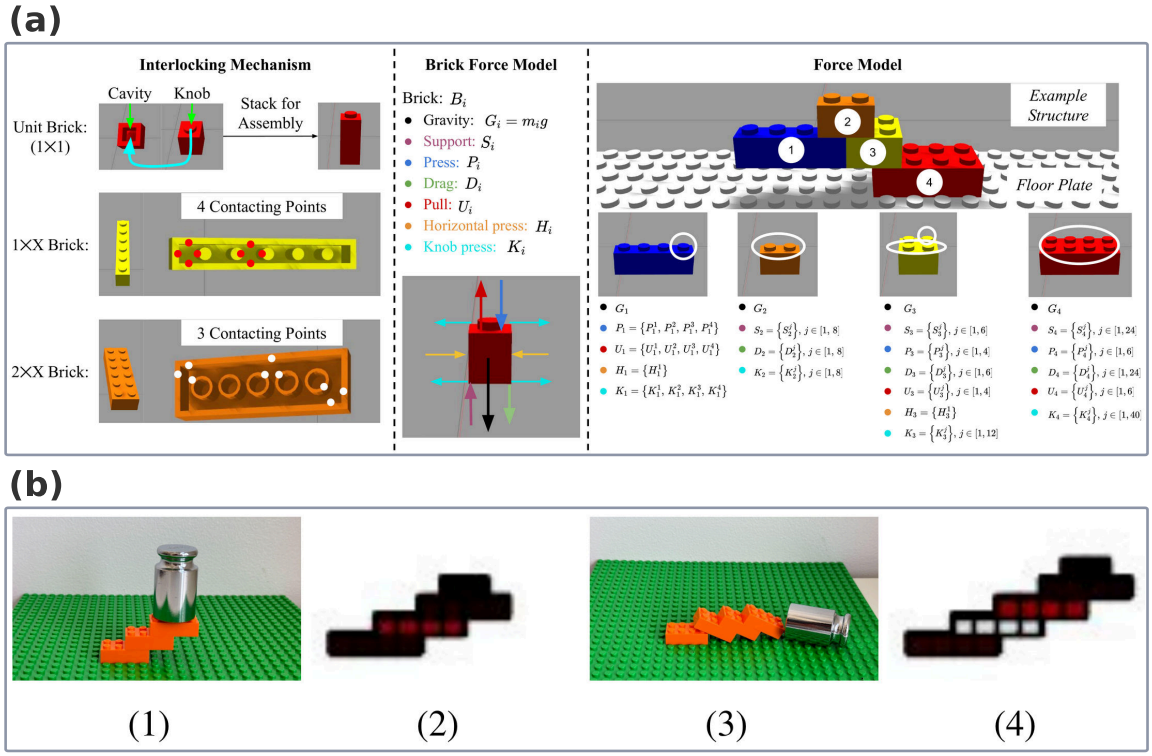


Figure 3.3: StableLego, the model this section adapts. **(a)** the per-brick force model, with three to four knob contacts per connection. **(b)** the same model run under an external load, localizing instability to individual bricks (black is low internal stress, red high, white collapsing). *Images: [26].*

3.1.3 Preliminaries and Notation

A structure is a set of blocks $\{B_i\}$ on the cubic lattice, where each block occupies one or more voxels and has mass m_i with weight $G_i = m_i g$ at its center of mass. Blocks interact through three interface types: *vertical* interfaces with the blocks above and below (through the center interlock), *lateral* interfaces with same-layer neighbors, and the *ground*. Table 3.2 collects the force variables. Figure 3.5 shows them on a single voxel, and Figure 3.4 shows the assembled system on a benchmark structure.

The decisive geometric difference from LEGO is not how many contacts a connection has but how far apart they sit. StableLego resolves a connection at three to four knob points spread over the brick's whole footprint, tens of millimeters apart, so every connection carries a moment-resisting couple of its own. The MILAbot vertical interface is resolved at three points as well, a triangle at cell offsets $(\frac{1}{8}, \pm\frac{1}{8})$ and $(-\frac{1}{4}, 0)$ that sits entirely inside the central interlock, some 28 mm across on the 75 mm lattice. Over that span the lever arms are short and the couple correspondingly weak, so the

	StableLego [26]	**This work**
Parts	LEGO bricks	voxel blocks, 75 mm lattice
Contacts	3–4 knobs, brick-wide	3 points inside the interlock, plus edge clips
Moment	couple within a connection	clips and neighbors, not the interlock
Tensile capacity	checked after the solve	finite, $T_{\text{snap}}^{\max}$, per clip
Solver	residual minimization	the same, extended
Loads	static, optional	the robots, through posed feet
States	the finished design	every intermediate state
Output	stability and stress per brick	verdict per block, margin per connection
Role	analysis	oracle inside the planner

Table 3.1: The stability model of this section against StableLego, the model it adapts. Above the rule, the differences follow from the voxel's geometry, while below it they follow from what the verdict is for. StableLego answers whether a finished design stands, whereas this model answers whether a fleet can build one.

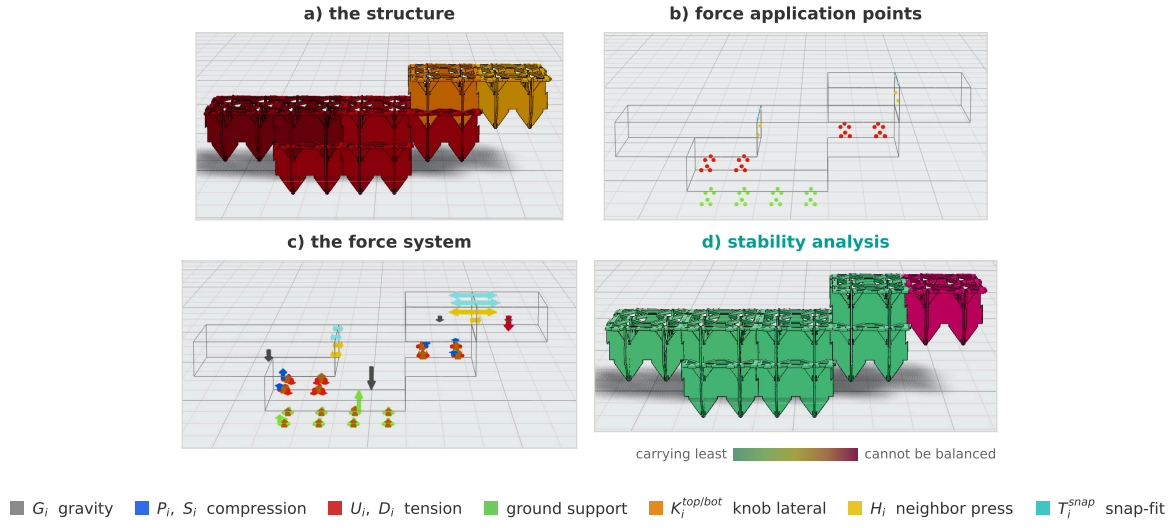


Figure 3.4: The force model on a benchmark structure, as the analyzer shows it. **(a)** the target, colored one hue per placed block (the colors are not stress). **(b)** the contacts at which the forces of Table 3.2 attach. **(c)** the full force system assembled at every block. **(d)** the per-block equilibrium verdict returned by the existence optimization, green stable and magenta unstable.

moment that keeps an overhanging block from rotating off comes instead from the clips and the neighbor presses. The model must therefore track where each force applies, not merely how large it is.

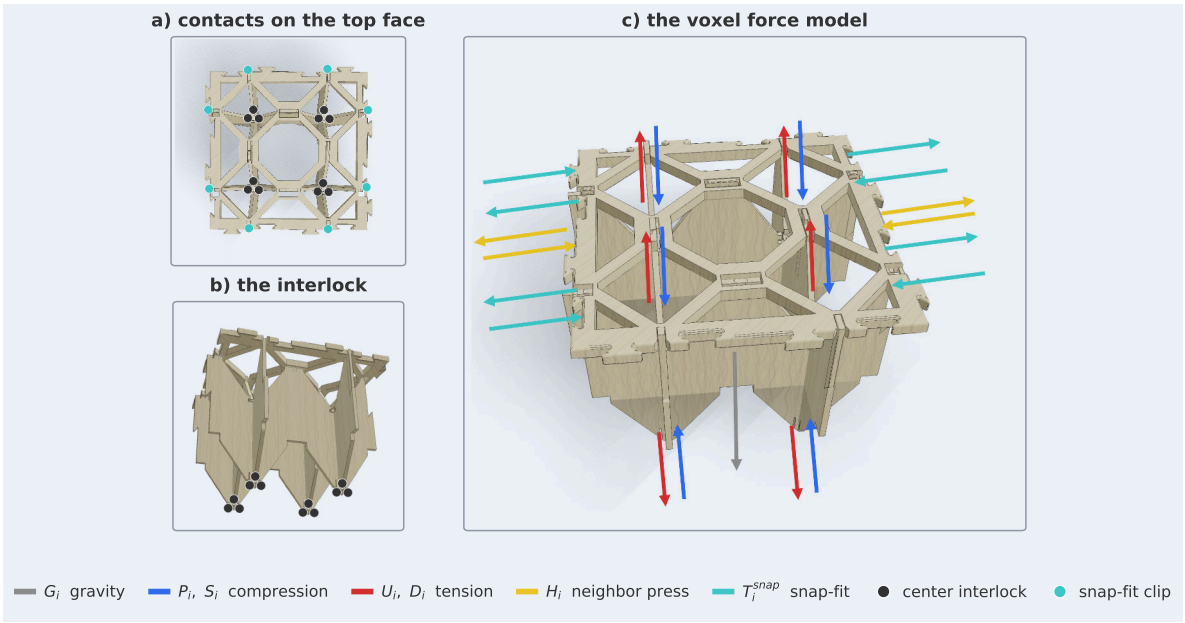


Figure 3.5: The center-contact voxel force model, on the fabricated 2 × 2 voxel. **(a)** the top face, with the center-interlock contacts drawn black and the snap-fit clips at the sides teal. **(b)** the underside prong triplets that engage the voxel below. **(c)** the force system on one voxel, with press and pull P_i, U_i, support and drag S_i, D_i, neighbor presses H_i, snap-fit forces T_i^{snap} of finite capacity $T_{\text{snap}}^{\max}$, and gravity G_i, each at the location given in Table 3.2. Lateral blocking K_i acts at the same interlocks (not drawn).

Symbol	Location	Meaning
G_i	center of mass	gravity
P_i	top contacts	pressing force from the block above (down)
U_i	top contacts	pulling force from a tight connection (up)
S_i	bottom contacts	supporting force from below (up)
D_i	bottom contacts	dragging force from the connection (down)
$K_i^{top/bot}(d)$	interlock contacts	lateral mechanical blocking of the interlock, $d \in \{x\pm, y\pm\}$
$T_i^{snap}(d)$	top-face edges	snap-fit forces, capacity-limited
$H_i^{(d)}$	side interfaces	normal press from same-layer neighbors

Table 3.2: Force variables of the center-contact snap-fit model.

3.1.4 Constraints

Equilibrium

Each block must be in static equilibrium. With $\mathscr{D} = \{x+, x-, y+, y-\}$:

$$G_i + (P_i + U_i + S_i + D_i) + \sum_{d \in \mathscr{D}} \left(K_i^{top}(d) + K_i^{bot}(d) + T_i^{snap}(d) + H_i^{(d)} \right) = 0 \tag{3.2}$$

$$L_i^G \times G_i + \sum_{F \in \mathscr{F}_i} L_i^F \times F = 0 \tag{3.3}$$

where L_i^F is the lever arm of force F relative to the center of mass of B_i, and $\mathscr{F}_i$ collects all contact forces of Table 3.2. Because vertical forces act at the contact centers, their lever arms are purely vertical, and the moment balance is carried by K_i, T_i^{snap}, and H_i.

Contact coupling

Forces match across shared interfaces. For B_j stacked on B_i:

$$P_i = S_j, \qquad U_i = D_j, \qquad K_i^{top}(x\pm) = K_j^{bot}(x\mp), \quad \text{etc.} \tag{3.4}$$

and for lateral neighbors, $H_i^{(x+)} = H_j^{(x-)}$.

Complementarity and admissibility

A contact cannot press and pull simultaneously:

$$P_i \cdot U_i = 0, \qquad S_i \cdot D_i = 0, \tag{3.5}$$

and all contact force magnitudes are non-negative. The lateral blocking forces K_i are unconstrained in magnitude. Like StableLego's knob forces, the interlock is mechanical blocking rather than friction, and failure of the part itself is excluded by assumption.

Snap-fit capacity

The snap-fits are small cantilevered features with a mechanical failure limit, and this limit is the key new parameter of the model:

$$\left| T_i^{snap}(x+) + T_i^{snap}(x-) \right| \le T_{\text{snap}}^{\max}, \qquad \left| T_i^{snap}(y+) + T_i^{snap}(y-) \right| \le T_{\text{snap}}^{\max}. \tag{3.6}$$

A structure whose equilibrium requires more snap-fit force than $T_{\text{snap}}^{\max}$ is declared unstable even if all vertical load paths are sound. StableLego cannot express this failure mode, which is the one that governs overhangs assembled from snap-fit voxels.

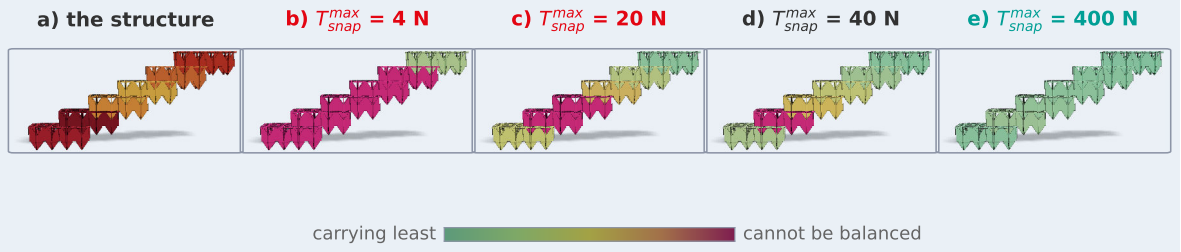


Figure 3.6: Sweeping the snap-fit capacity $T_{\text{snap}}^{\max}$ on the staircase benchmark. **(a)** the structure, colored one hue per block, while the panels that follow recolor the same six by verdict. At 4 N five of six are unstable **(b)**, at 20 N two remain **(c)**, at the measured 40 N all six are stable with the snap-fits running close to capacity **(d)**, and at 400 N the structure is comfortably stable **(e)**.

The measured capacity of the fabricated voxel is $T_{\text{snap}}^{\max} \approx 40\,\text{N}$, and the parameter is decisive. Sweeping it on a staircase benchmark (Figure 3.6) moves the same six blocks from five unstable at 4 N to fully stable at the measured 40 N. Repeating the sweep with a robot posed on the top step (Figure 3.7) pushes that same 40 N structure back to four of six blocks unstable, the effect that the robot-load injection of Section 3.1.5 accounts for in every verdict.

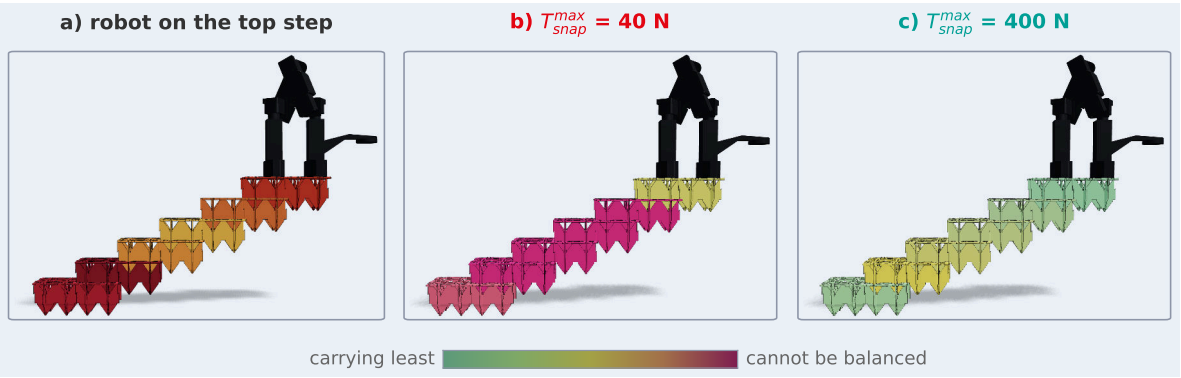


Figure 3.7: The same staircase, now carrying a posed MILAbot on its top step. **(a)** the loaded state, before any verdict. **(b)** at $T_{\text{snap}}^{\max} = 40\,\text{N}$, the measured capacity and enough for the bare structure, the robot's own weight leaves four of six blocks unstable. **(c)** at 400 N the loaded staircase is stable throughout.

3.1.5 Robot Load Injection

During construction the structure also carries its builders, since each posed robot contributes weight $W = m_r g$ through its two feet, each foot a 2×1 stripe of top-layer cells. The split between feet projects the robot's center of mass onto the segment joining the foot centroids:

$$t = \mathrm{clamp}\left(\frac{(c - \ell_1) \cdot (\ell_2 - \ell_1)}{\|\ell_2 - \ell_1\|^2}, 0, 1 \right), \qquad F_{\ell_2} = W t, \quad F_{\ell_1} = W(1 - t), \tag{3.7}$$

with each foot's share distributed evenly over its cells and entering the force and torque balances of the blocks below as external downward loads. If the center of mass is unknown the split defaults to 50/50. Blocks that stand on their own then fail under the robot's weight, the dominant load in every assembly state. Stability is therefore assembly-state-dependent, which motivates checking every step of a build sequence rather than only the finished structure.

3.1.6 Solving the Force Balance

Following StableLego, the existence question is posed as a continuous optimization over the force variables, solved in Gurobi with the settings of Appendix A:

$$\min \sum_i (\|\Sigma F_i\|_1 + \|\Sigma \tau_i\|_1) \; + \; \alpha \sum_i \max(S_i, P_i) \; + \; \beta \sum_i \Sigma_F F \tag{3.8}$$

The first term drives residual force and torque imbalances to zero, so a solution with zero residual certifies equilibrium, while a strictly positive residual localizes instability to the blocks that cannot be balanced. The α term penalizes excessive vertical loading (yielding the per-block stress visualization), and β regularizes toward minimal force solutions. Figure 3.8 shows the resulting verdicts across the benchmark suite, where every structure solves in one to five seconds.

3.2 Making Structures Stable

Figure 3.8 has already answered that question for the structures this thesis works with. When they are patterned layer by layer, the way the previous pipeline does it, four of the five benchmarks carry blocks that the analysis rejects, up to 336 of 1 568 on the covered shelter, while only the voxel cart comes back clean. The reasons are always the same, namely that vertical seams align, that columns carry no lateral ties, and that overhanging blocks demand more snap-fit force than the parts can supply.

Given these failures, two remedies are available. Either the target is kept and material is added until it stands, or the same target is laid out differently so that it stands on its own. Both are developed here, in the order they were built, and both fall short.

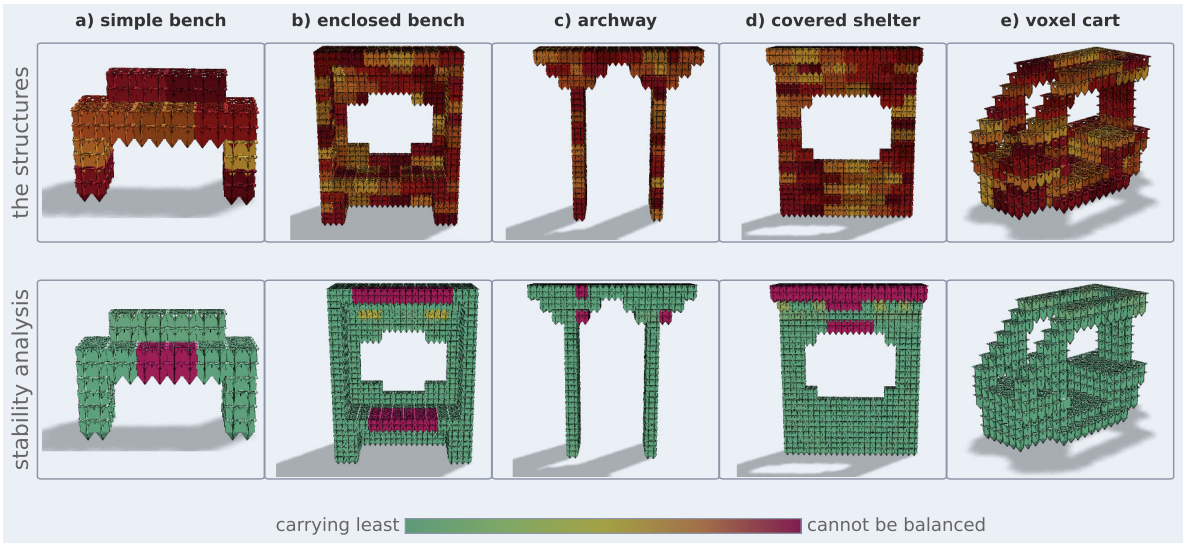


Figure 3.8: Per-block verdicts across the benchmark suite. Top row: block identities. Bottom row: verdicts. **(a)** the simple bench, voxelized layer by layer by the previous pipeline, is the baseline, with 16 of 192 voxels unstable. **(b)** the enclosed bench flags 272 of 2,992 (4.6 s). **(c)** the archway, 112 of 1,728 (2.7 s). **(d)** the covered shelter, 336 of 1,568 (2.7 s). **(e)** the voxel cart is fully stable, 808 of 808 (1.2 s).

3.2.1 Support-Based Stabilization

The first remedy mirrors additive manufacturing, adding sacrificial support material wherever the structure cannot hold itself.

Manual supports

The analyzer lets the user place support blocks, typically columns under unstable overhangs, and re-run the analysis interactively. This is effective and transparent, but it remains a human-in-the-loop patch rather than a pipeline stage, and it encodes no reusable knowledge.

Smart tree supports

The automated variant generates supports the way slicers grow trees. Starting from each unstable block, the algorithm searches a path downward to ground or to a stable region, adds support blocks along it, re-runs the stability analysis, and iterates until the verdict clears. Paths can branch and merge, so clustered overhangs share one trunk (Figure 3.9).

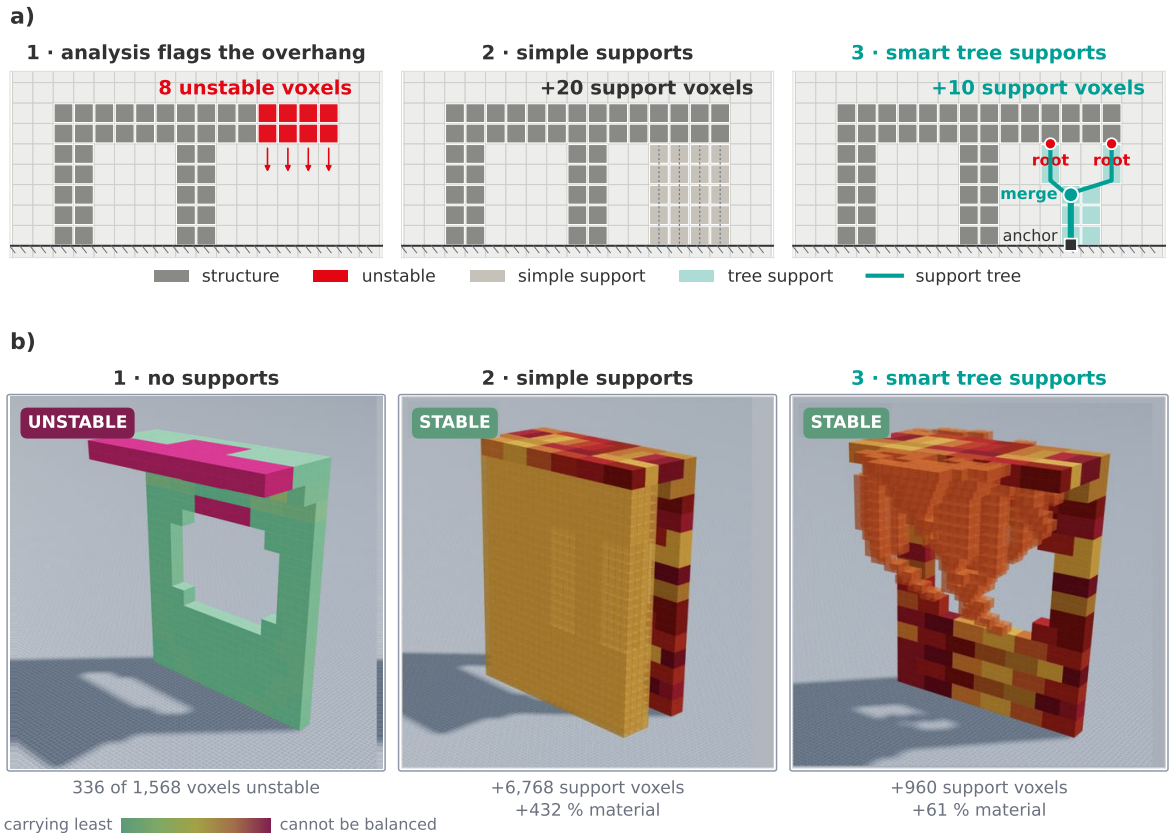


Figure 3.9: The two support strategies, side views on the voxel grid. **(a)** the mechanism. **1** the stability analysis of Section 3.1 flags the voxels at the free end of the overhang. **2** the simple remedy drops one column straight to ground under each of them. **3** the tree variant merges branches into a single shared trunk anchored to ground. Voxel counts are for the schematic. **(b)** the same three columns on the hardest benchmark, the covered shelter, on the scale of Figures 3.6 and 3.8. Simple columns reach a stable verdict at +432 % of the structure's own material, the tree variant at +61 %, which is seven times less.

What both variants cost on the full benchmark suite is measured in Section 6.1.

Limits of supports

Supports do work, in the sense that every benchmark structure can be stabilized this way, but they treat symptoms rather than causes, and three costs accumulate. The first is *material and time*, as every support voxel is placed by a robot, so that on tall overhangs the overhead grows faster than the structure itself. The second is *removal*, which unlike 3D-printing support is itself a robotic disassembly problem with its own stability constraints. The third is *architectural*, in that stability is

bolted on after pattern detection rather than designed into the structure.

3.2.2 Stability Through Voxel Patterning

The second remedy attacks the pattern itself, generating arrangements whose seams interlock, namely masonry bonds discovered by optimization.

ILP formulation

Pattern assignment is posed as an exact cover problem, in which every target voxel must be covered by exactly one catalog block. The objective rewards *staggering*, giving bonuses to blocks whose seams alternate between layers and penalties to vertically aligned joints, which pushes solutions toward running-bond arrangements that transfer load laterally.

Achievements and limits

On the benchmark suite the ILP patterns pass full-structure stability analysis almost universally. How far it beats greedy layer-by-layer patterning, and where it still fails, is measured in Section 6.1. Three hard limits nevertheless emerged from this formulation:

1. **Scale.** The ILP grows combinatorially with structure volume, so solve times that are seconds on the simple bench become hours at architectural scale.
2. **No learning.** The staggering objective is hand-tuned, encoding our intuition about good patterns rather than evidence, so that every new block type or constraint means re-tuning weights.
3. **Stable $\neq$ buildable.** ILP certifies only the *finished* structure. The intermediate states are unconstrained, and in practice ILP patterns frequently require placement orders that pass through unstable configurations, or demand placements the robot cannot reach.

3.2.3 Requirements for Stability-Aware Planning

Taken together, these failures yield the specification for Section 3.3. The mechanism must (i) constrain *every intermediate state*, which couples pattern choice to placement order and makes the problem sequential. It must (ii) respect *robot constraints*, meaning placement from above, reachability and standing positions, so that its output is executable and not merely stable. It must (iii) *scale*, which rules out per-state ILP and per-action exact solves and demands cheap feasibility

checks. Finally, it must (iv) *learn*, so that pattern quality comes from optimization pressure rather than hand-tuned weights. These four requirements lead directly to the reinforcement-learning formulation developed next.

3.3 Stability-Aware Sequence Planning

This section presents the planner that meets the requirements established in Section 3.2, starting from a definition of what has to be guaranteed. Targets that admit no buildable sequence at all are handled by the target-evolution loop of Section 3.4.

3.3.1 Buildability

> *A structure is **buildable** under a robot setup if there exists a placement sequence such that (i) every prefix of the sequence is statically stable, and (ii) every placement in the sequence is reachable by a robot from a valid stance, which on this system means the block can be brought down onto its seat from above.*

Two things sit deliberately outside this definition. The robots' own weight belongs to the stability verdict of Section 3.1, which the planner queries rather than guarantees, and the fine detail of the arm trajectory belongs to the twin, which checks the gesture afterwards.

Buildability is strictly stronger than stability. A bridge is sound as a closed arch but passes through unsupported cantilevers on the way there, and a block whose seat is walled in by blocks placed earlier is unplaceable. Buildability is also *setup-dependent*, changing with where the robots stand and with the support strategy.

The pipeline implements this definition directly as a *buildability checker*. Given a structure, a sequence and a robot setup, the checker replays the build in the twin and returns either `YES` or the first failing step with its cause.

3.3.2 Related Work: Assembly Sequence Planning

Deciding in what order to place parts, known as assembly sequence planning (ASP), is NP-hard in general. Physics-based planners such as Assemble Them All sidestep combinatorial search by planning disassembly in simulation and reversing it, generalizing across thousands of part geometries [31], and ASAP carries that line to gravity, searching over disassembly for sequences in which every sub-assembly is stable on a support surface [32]. Both test rigid-body feasibility on CAD

parts in an arm workcell rather than a connection-level force balance over a lattice. Reinforcement learning entered architectural assembly with Learn2Assemble, which combined structured representations and tree search to place blocks toward target silhouettes [33].

The work this section extends is the physics-aware combinatorial ASP of Liu et al., in which a reinforcement learning policy proposes placements while a data-free action mask, backed by a stability solver, removes actions that would create unstable intermediate states (Figure 3.10) [34]. The learning machinery builds on proximal policy optimization [35] with invalid-action masking [36]. On LEGO benchmarks this achieves 100 % success where unmasked baselines fail, but at small scale, on 48^3 grids of flat bricks. Architectural voxel assembly instead requires action spaces orders of magnitude larger, compound multi-voxel blocks, robot placement constraints, and a mask cheap enough to evaluate millions of times. Table 3.3 sets the two side by side.

	Physics-aware ASP [34]	**This work**
Parts	LEGO bricks, from an inventory	voxel blocks, compound and pattern
Action	brick, position, orientation	the same, or a bonded stagger chain
Mask tests	support, inventory, stability	support, stability, placement from above, tileable remainder
Stability	simplified LEGO force model	the model of Section 3.1
Guarantee	stable intermediate states	buildable: stable *and* reachable
Executed by	a bimanual arm cell	a fleet climbing the structure
Workspace	a 48×48 plate	16^3 patches, tiled to full targets
Action space	$H \times W \times D \times N \times 2$, over a million	target-valid placements only, hundreds
Mask cost	one stability solve per action	precomputed tables, compiled evaluation
Reuse	none	symmetry detection and warm starts across patches
Reward	coverage	coverage, layer completion, staggering
Policy	MaskablePPO	MaskablePPO, 3D-CNN with per-layer attention

Table 3.3: The sequence planner of this section against the physics-aware ASP it extends. Above the rule is the problem each planner is given, below it the machinery that moves the method from a plate to a building.

3.3.3 Planning as Reinforcement Learning

Formulation

Sequence planning is cast as a Markov decision process. The *state* is the pair (current occupancy, target occupancy) on the workspace grid, encoded as 3D tensors. An *action* places one catalog block,

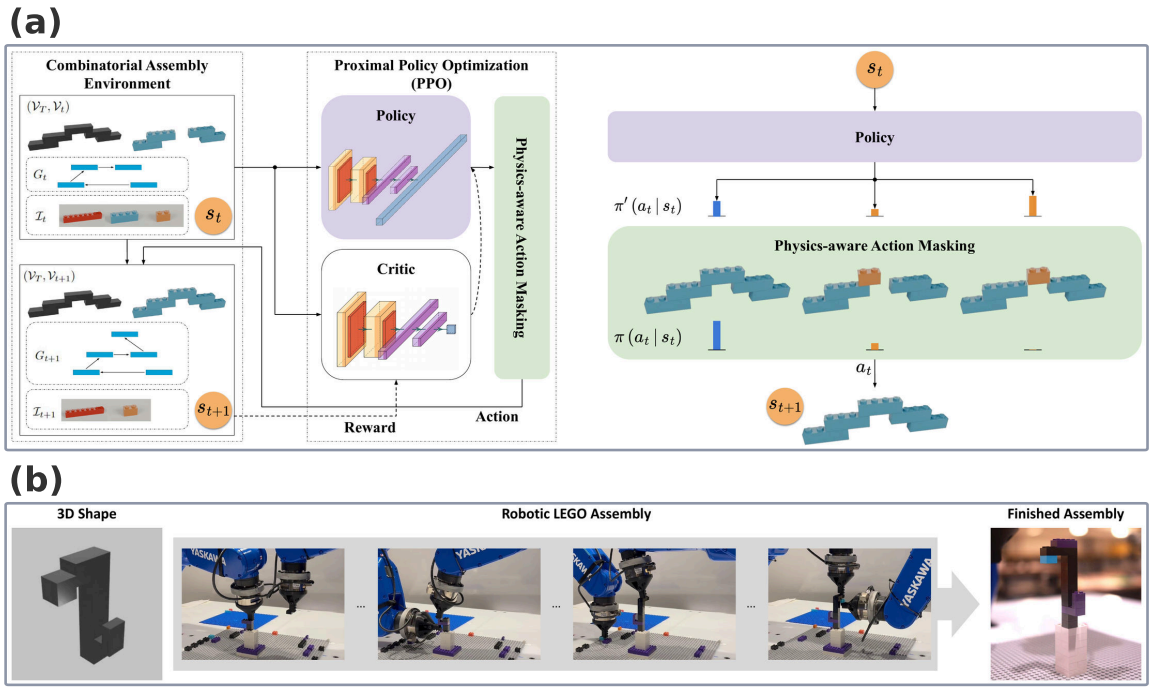


Figure 3.10: Physics-aware combinatorial ASP, the planner this section scales up. **(a)** the masked policy proposing placements on a LEGO plate. **(b)** the resulting sequence executed by a bimanual robotic system. *Images: [34].*

specified by its type, orientation and position, and Figure 3.13 illustrates one such decision on our voxels.

The *reward* combines target progress with structural shaping, namely layer-completion bonuses and rewards for staggered, interlocking placements (Figure 3.11). The policy is a MaskablePPO agent [35, 36] with a 3D-CNN feature extractor, following the physics-aware ASP architecture [34]. Figure 3.12 shows that architecture in training and at deployment.

Physics in the action mask

Before each decision, infeasible actions are removed: placements outside the target, placements without support below, placements that collide with the robot's required approach (nothing may already occupy the column above a placement cell), and placements whose resulting state fails the stability screen. A final condition masks placements that would strand a residual volume no catalog block can tile. The policy can therefore *only* choose among placements that keep the structure sound, so that stability is guaranteed by construction rather than encouraged by reward.

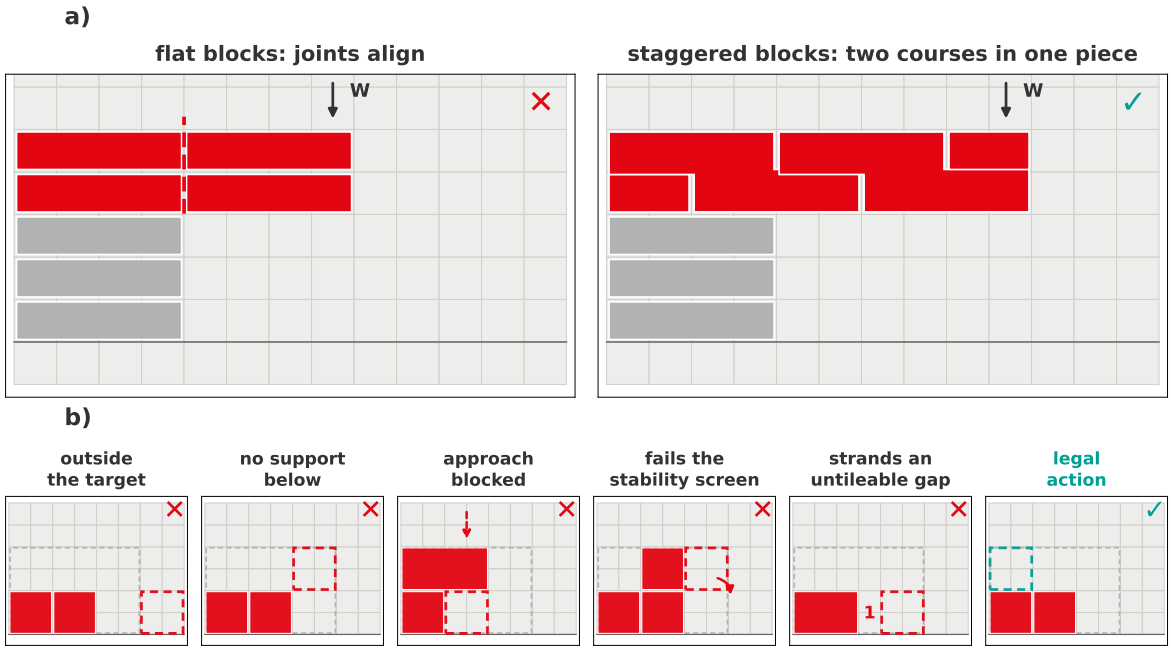


Figure 3.11: Reward shaping and mask conditions. **Top:** why the reward favors staggered blocks. Flat blocks stack into separate columns that share one continuous joint plane, which must carry an overhang in snap-fit tension alone (left). A staggered block instead joins two offset courses into one rigid piece, so tip load returns to the support through compression on the overlaps (right). **Bottom:** the five mask conditions, each on a small build state (side view, gray dashed line = target region, red = placed blocks, dashed = candidate placement).

The planner of Liu et al. pays for this guarantee with an exact solve per masked action, which is prohibitive beyond LEGO scale [34]. We therefore replace the per-action solver with *geometric mask heuristics*, namely support and interlock conditions derived from the model of Section 3.1 and evaluated in microseconds. The full optimization is reserved for the buildability checker that certifies the finished sequence.

Observability improvements

The observation of Liu et al. [34] still starves the policy of long-horizon information. We therefore add per-column height maps, layer-completion progress, remaining-region indicators, and an active construction layer hint with z-attention. A second addition prevents dead ends. *Invalid-residual detection* simulates each candidate placement and verifies that the remaining empty target volume is still tileable by the available block catalog.

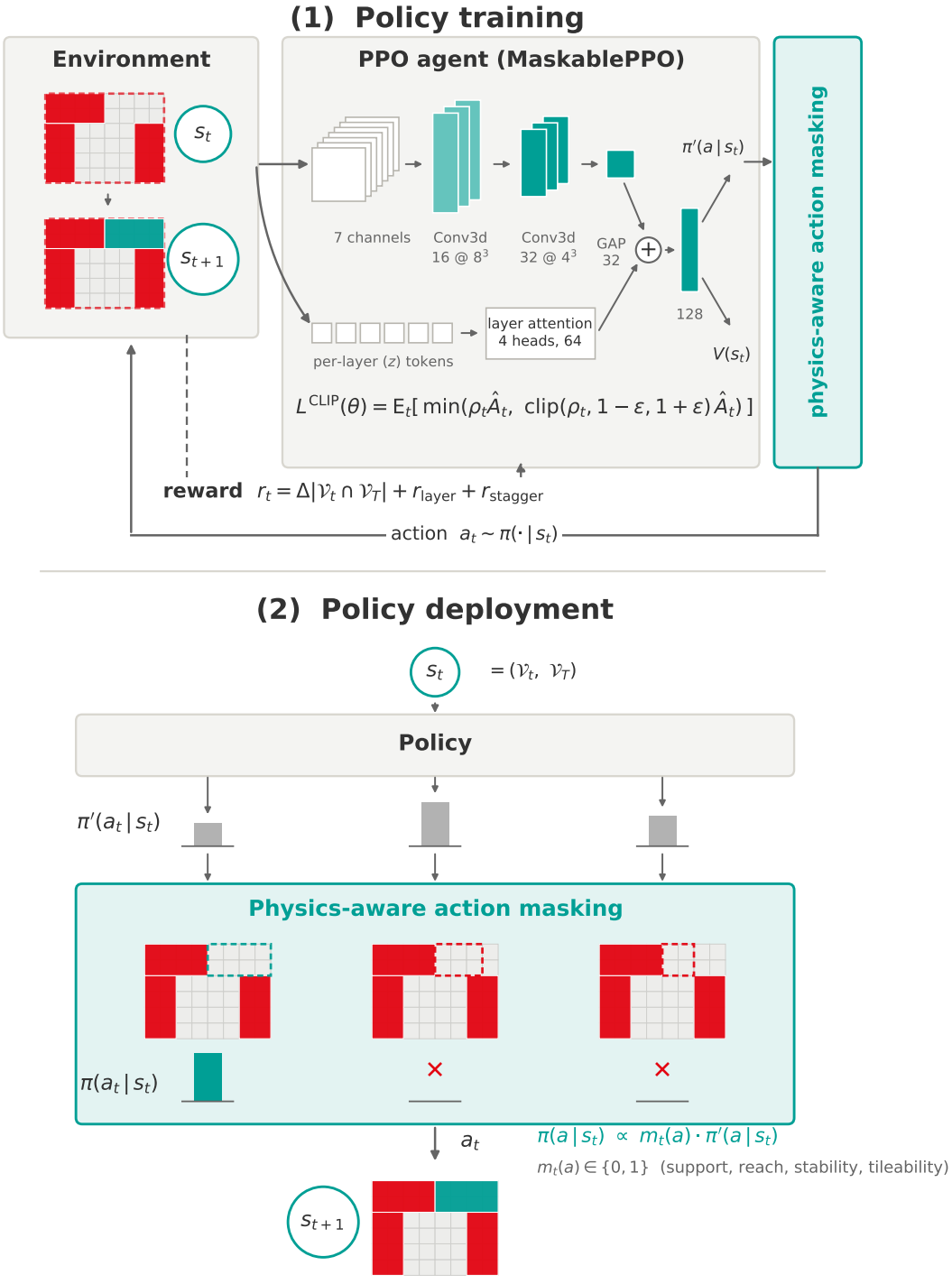


Figure 3.12: The sequence planner in training (top) and at deployment (bottom). A 3D-CNN with per-layer attention encodes current and target occupancy into a shared feature vector for the policy and critic heads. The physics-aware mask then zeroes infeasible placements, so the sampled distribution $\pi(a \mid s_t) \propto m_t(a)\,\pi'(a \mid s_t)$ contains only sound placements.

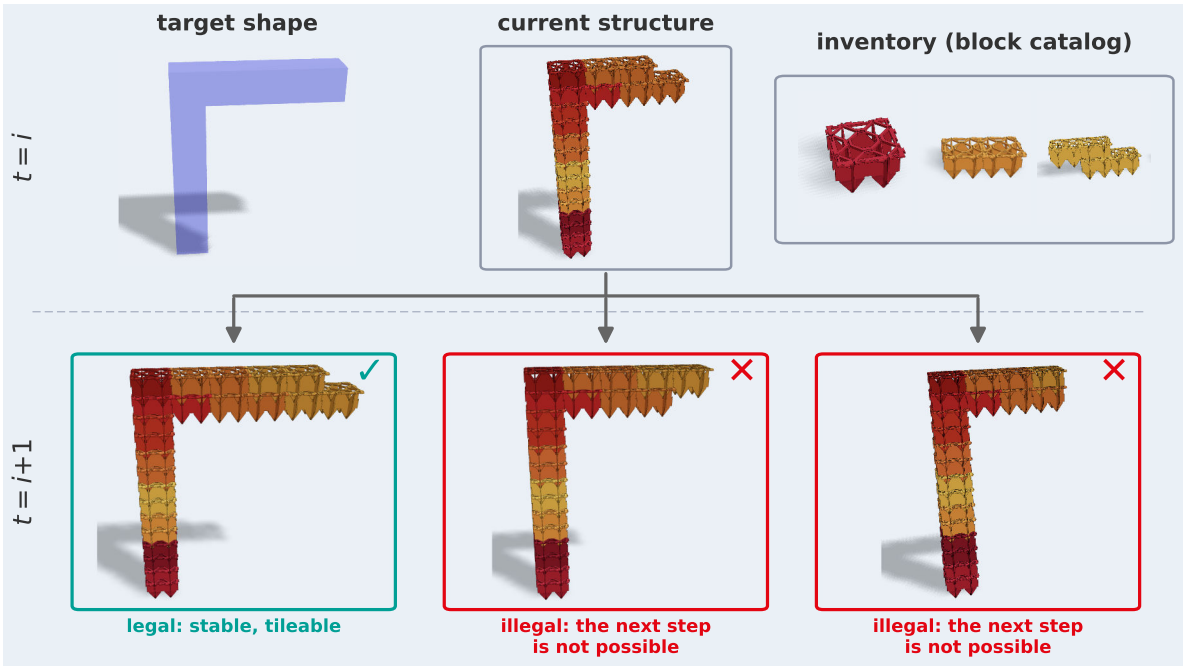


Figure 3.13: The combinatorial assembly decision on MILAbot voxels (after [34]). At step $t = i$ the planner sees the target shape, the current structure, and the block inventory. The mask removes the illegal candidates at $t = i+1$ before the policy chooses.

3.3.4 Scaling to Architectural Structures

In order to close the gap from 48^3 LEGO benchmarks to architectural volumes, three mechanisms were implemented.

Patch decomposition. The policy is trained on local 16^3 workspaces. Large targets are decomposed into patches and planned patch by patch with shared boundary conditions, while symmetry detection mirrors and reuses solved patches across repetitive structures. A warm-start curriculum, pretraining on a single canonical structure before patch training, accelerates convergence on every subsequent target.

Macro placements. One action may place a bonded chain of staggered blocks rather than a single block. Long stagger chains therefore collapse from many decisions into one, shortening the horizon and biasing structure quality.

Action-space reduction. Only target-valid placements are enumerated, via precomputed placement tables, and mask evaluation is compiled (Numba/JAX). Together these collapse the space the policy chooses among from every conceivable placement in the workspace to the handful that are sound by construction.

The combined effect on the benchmark suite is that training is cheap enough to be interactive, and emergent staggered patterns appear without any ILP in the loop. Figure 3.14 shows both halves of that claim, on one cantilever end to end and then across the benchmark suite. That cantilever returns in the hardware experiments of Section 5.3.3. Section 6.2 measures all three claims: the funnel these mechanisms produce, the stability of the resulting patterns against the hand-designed strategies, and what a plan costs to train.

3.4 Target Evolution and Repair

The planner of Section 3.3 guarantees stability for every sequence it produces, but some targets admit no buildable sequence at all. The MILAbot builds bottom-up and places only from above, so overhangs float until their span closes, and layers whose area cannot be tiled by the even-area block catalog leave residues no policy can fill. On such targets the agent does not fail loudly, but instead *dead-ends* mid-build, with no legal action left. When the sequence cannot be fixed, this section therefore fixes the target instead. A geometry that is not buildable is grown, minimally and purely additively, into one that is, and the original, untouched planner then fills it. Because the grown shape *contains* the original, filling it completely covers the original completely.

Changing the target instead of the plan is an established idea. Funes and Pollack showed decades ago that buildable structures can be evolved under physical feasibility constraints, letting the search discover geometry that a fixed designer would not have proposed [37]. The same stability analysis this thesis builds on has since been pushed upstream into generation itself. Indeed, BrickGPT emits brick structures from a text prompt with an autoregressive model and prunes infeasible continuations at inference through a physics-aware rollback [38]. This section takes the narrower, deterministic route, in which the target is not regenerated but *repaired*.

3.4.1 Why Targets Fail

A target is buildable by the planner if and only if every layer can be tiled by the brick set and every cell is supported from below. Non-buildability therefore has recurring, recognizable causes on the voxel lattice:

- **Floating bases:** a column has a lowest occupied cell that does not reach the ground plane.
- **Odd solid layers:** a filled layer has an odd span, a 5×7 deck for instance, which no set of disjoint 2×2 footprints can cover completely.
- **Single-layer overhangs:** roof-exposed cells have nothing in the cell below, forming a one-course cantilever that has nothing to stand on during construction even though the finished shape is sound.

Figure 3.14: Planned sequences, from one structure to the whole suite. **(a)** the cantilever end to end. The planned sequence step by step, A–H, with the column rising first and the deck then extending outward (**1**), the finished structure at the measured capacity $T_{\text{snap}}^{\max} = 40\,\text{N}$ with every block stable (**2**), and the same cantilever assembled from fabricated voxels (**3**). **(b)** planned sequences on four benchmark structures, one row each, sampled at five stages (A–E) with the stability verdict at right. Panels within a row share one crop window and one ground line.

- **Thin ribbons:** a run is one voxel wide, with no same-layer neighbour on one axis, so that no catalog block fits it.
- **Isolated cells:** a cell forms a 1×1 island on its layer, with zero same-layer neighbours.
- **Enclosed voids:** a pocket inside the footprint is walled in by its own neighbours, so that it can no longer be reached once the cells around it are down, since MILAbot places only from above.

Figure 3.15 shows all six on the lattice. The first five are not a taxonomy invented for the thesis, but the detectors implemented in `stability_evolve/validators.py`, drawn in the order the repair loop resolves them. The sixth is the one they miss, which is why detection also runs a probe build.

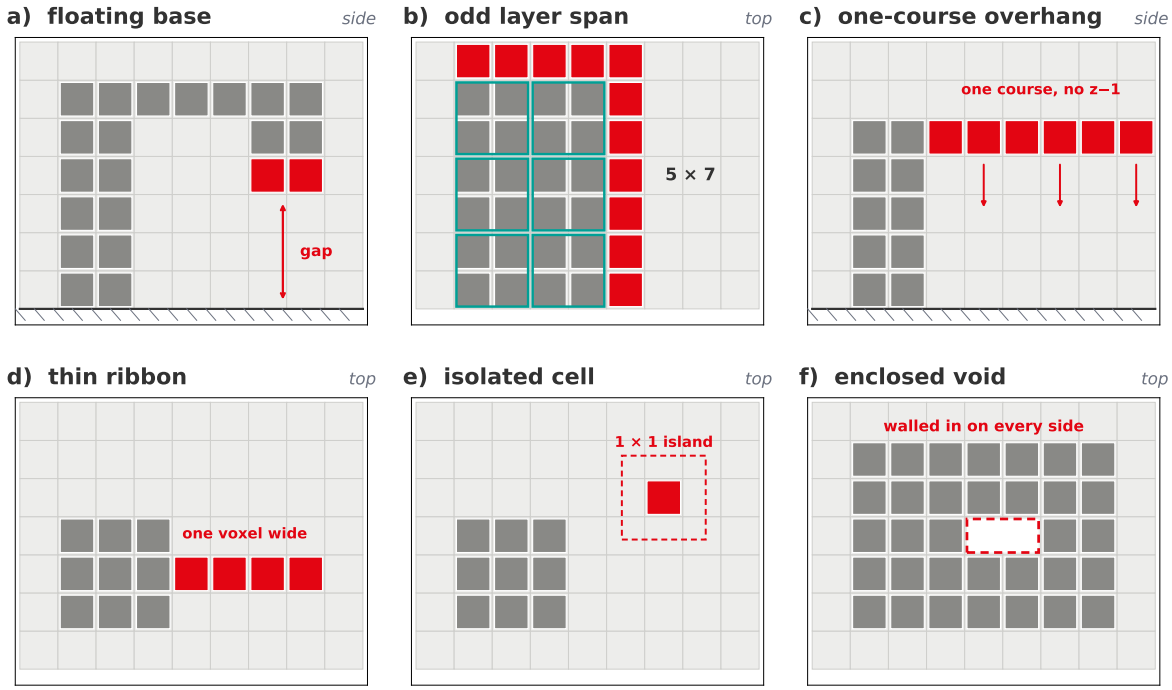


Figure 3.15: The six ways a target fails the buildability check, with the offending cells in red. **(a)** one leg of a portal stops short of the ground. **(b)** an odd span, where six 2×2 footprints (teal) fit into a 5×7 deck and eleven cells are left over, and no arrangement does better. **(c)** a one-course overhang with nothing at $z-1$ to stand on. **(d)** a run one voxel wide. **(e)** a 1×1 island. **(f)** a pocket walled in by its own neighbours. Panels (a) and (c) are side views, the rest top views of a single layer.

Figure 3.16: Target evolution as the pipeline runs it. **(a)** the loop, in which a buildable target goes straight to the planner with nothing added, while any other target is first grown into a buildable superset. **(b)** the same two steps on a one-layer cantilever. The planner dead-ends on the original target, since the deck has nothing to stand on and a one-voxel layer cannot be tiled. The grower is generous by design, offering three more courses under the deck (orange), and the planner keeps only what a buildable order requires, which here is one of the three, so the deck is built two courses thick (teal).

3.4.2 Detect, Grow, Build

The pipeline frontend exposes evolution as two buttons (Figure 3.16), and no other evolution mechanism is part of the pipeline.

Grow the target. *Evolve target* runs a deterministic repair loop, implemented in `stability_evolve`. *Validators* detect the specific violations of Section 3.4.1 and *mutations* fix them in priority order: support columns are dropped under floating volume, odd-area regions are padded to even, one-voxel ribbons are thickened, and unsupported roof overhangs receive support to a set depth. Every mutation is local, interpretable and cheap. Targets larger than one 16^3 workspace are grown chunk by chunk on the same grid the planner uses. The strict validator is conservative and the planner sometimes builds structures it rejects, so detection pairs the validator with a *probe build*, a cheap capped planning attempt that settles the question before any mutation

is applied.

Train and build on it. *Train ASP evolve* hands the grown target to the *unmodified* planner of Section 3.3, warm-started from the pretrained cantilever policy, so that the 3D-CNN feature extractor transfers while the per-target action head is retrained. The warm start is what lets large targets build quickly instead of dead-ending while learning from scratch. Coverage is graded on the *original* target, and the added volume is available as scaffold rather than required, so the built structure sits between the original and the grown superset.

The deterministic step handles the *geometry*, deciding where volume must be added, and the learned step handles the *sequence*, deciding in what order to place it. Neither needs to be modified for the other. What that costs, and how it behaves across the benchmark set, is measured in Section 6.3.

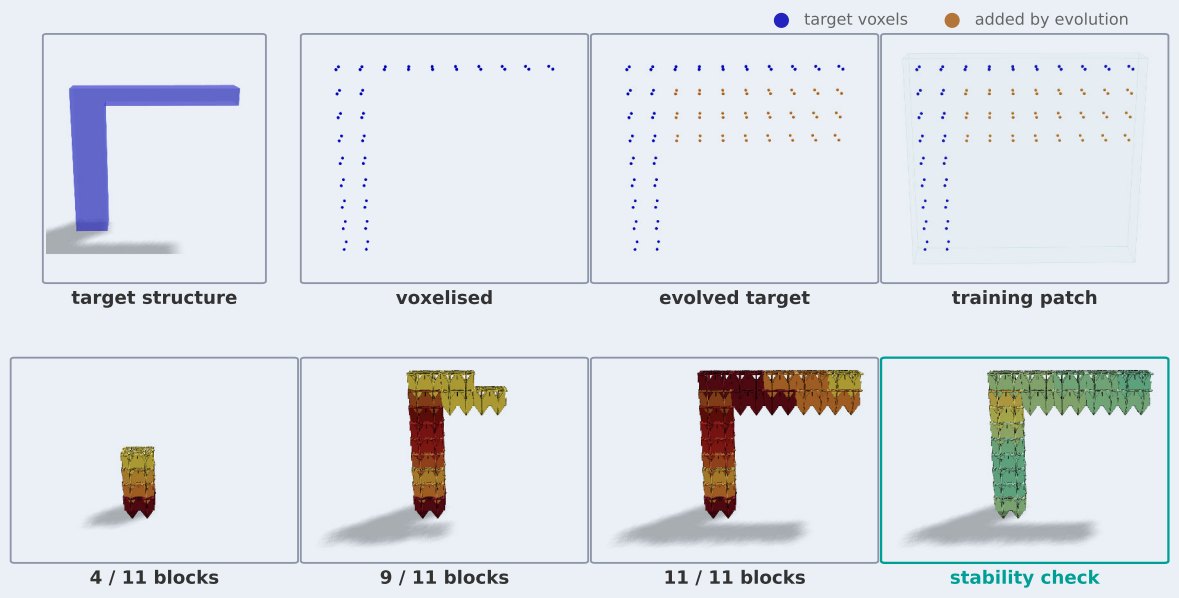


Figure 3.17: One target through the whole loop. Top: the input mesh, then its voxelisation (blue), then the evolved target in which growth offers candidate voxels (orange) to ground the overhanging arm, and finally the 16^3 training patch the policy sees around it. Bottom: the warm-started planner's sequence at three points in the build, and the stability verdict on the finished structure, on which every block passes. The sequence covers the original target using a fraction of the offered voxels. Panels within each row share one crop window, so the growth is read directly.

Figure 3.17 follows a single target through that whole loop, from the input mesh to a finished, verified structure. This is one run of the full pipeline rather than a measurement, and the pieces of this section compose with no step done by hand between them.

Chapter 4

Digital Twin Simulation

While a build order certified on the voxel grid guarantees that the structure stands, it says nothing about whether a robot can carry that order out. The MILAbot v2 generation changed the geometry, the gripper and the locomotion behaviors [25], and the simulator inherited from the previous pipeline was never brought back into agreement with the machine, so the placements it produced had no validated robot behavior behind them.

This chapter rebuilds that side of the system, presenting a full MILAbot v2 simulator that covers forward and inverse kinematics, locomotion primitives, stance-level path planning, block placement and multi-block build orchestration. It turns the abstract sequences of Sections 3.3 and 3.4 into motions a physical robot can perform, and it provides the foundation for the connected pipeline of Section 5.1 and the hardware link of Section 5.3. Its six tools are each exposed both as an interactive simulator page and as a library the pipeline calls, and they are presented below in the order they compose.

4.1 Robot Model and Forward Kinematics

The robot is modeled from a URDF built on the MILAbot v2 geometry [25], namely a 5-DOF serial arm mounted on a dual-foot lattice base. The world is metric, Z-up, and aligned to the 75 mm voxel lattice, so every kinematic quantity has a direct grid interpretation. The forward-kinematics tool exposes the model interactively, with joint sliders driving a live TCP pose, and reports the gripper position in voxel indices (g_x, g_y, g_z), which is the coordinate system in which every placement in the pipeline is expressed. The joint convention is fixed here and used throughout the thesis. Figure 4.1 names the five revolute joints $\theta_0 \ldots \theta_4$ and their positive senses, with the render and the schematic side by side.

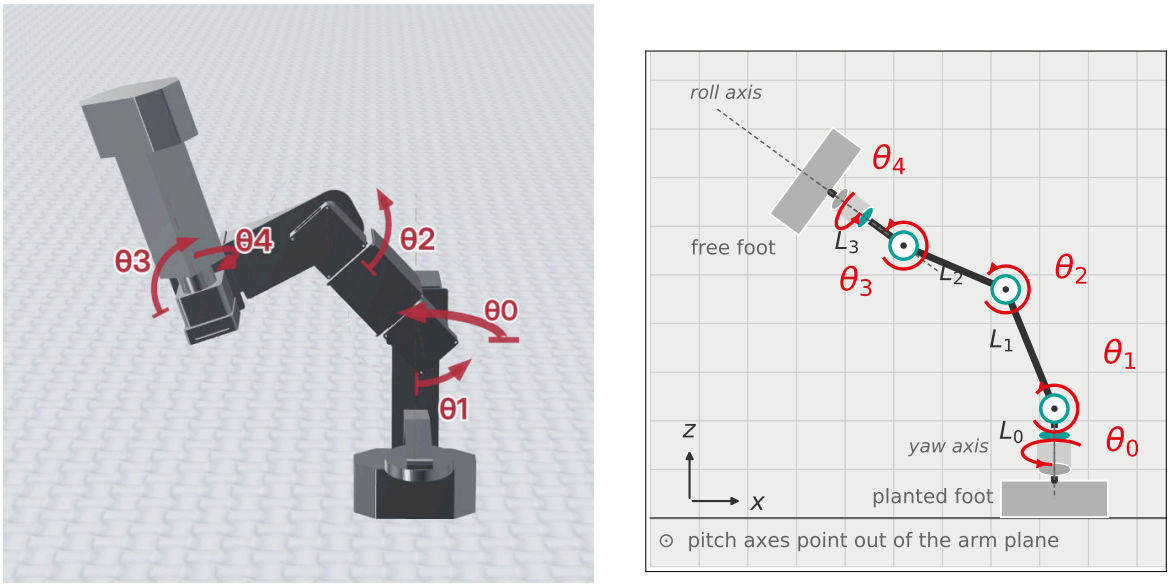


Figure 4.1: Joint convention of the 5-DOF arm: render (left) and kinematic schematic (right). Red arrows give the positive sense of each revolute joint: θ_0 rotates the arm about the vertical yaw axis of the base mount, θ_1, θ_2 and θ_3 are the pitch joints acting in the arm plane (axes out of the plane, $\odot$), and θ_4 rotates the gripper about the roll axis of the last link. Links L_0–L_3 connect consecutive joints.

4.2 Inverse Kinematics

Placement planning solves the inverse problem thousands of times per build, on a geometry the solver has never been tuned for, so robustness matters as much as accuracy. The core is damped least-squares over the full Jacobian, and four mechanisms sit around it, each added because a particular failure showed up in practice:

- **Dual-branch seeding:** every solve is attempted from elbow-up and elbow-down seeds, and the best valid branch is kept.
- **Adaptive damping with backtracking:** the damping factor adjusts to conditioning, and steps that increase the error are rolled back.
- **Warm starting with joint continuity:** consecutive solves seed from the previous solution with a joint-continuity cost, producing smooth motion along a gesture.
- **Perturbed multi-seed fallback:** when a solve stalls, randomized restarts around the seed recover it.

Figure 4.2 shows all four mechanisms on one target.

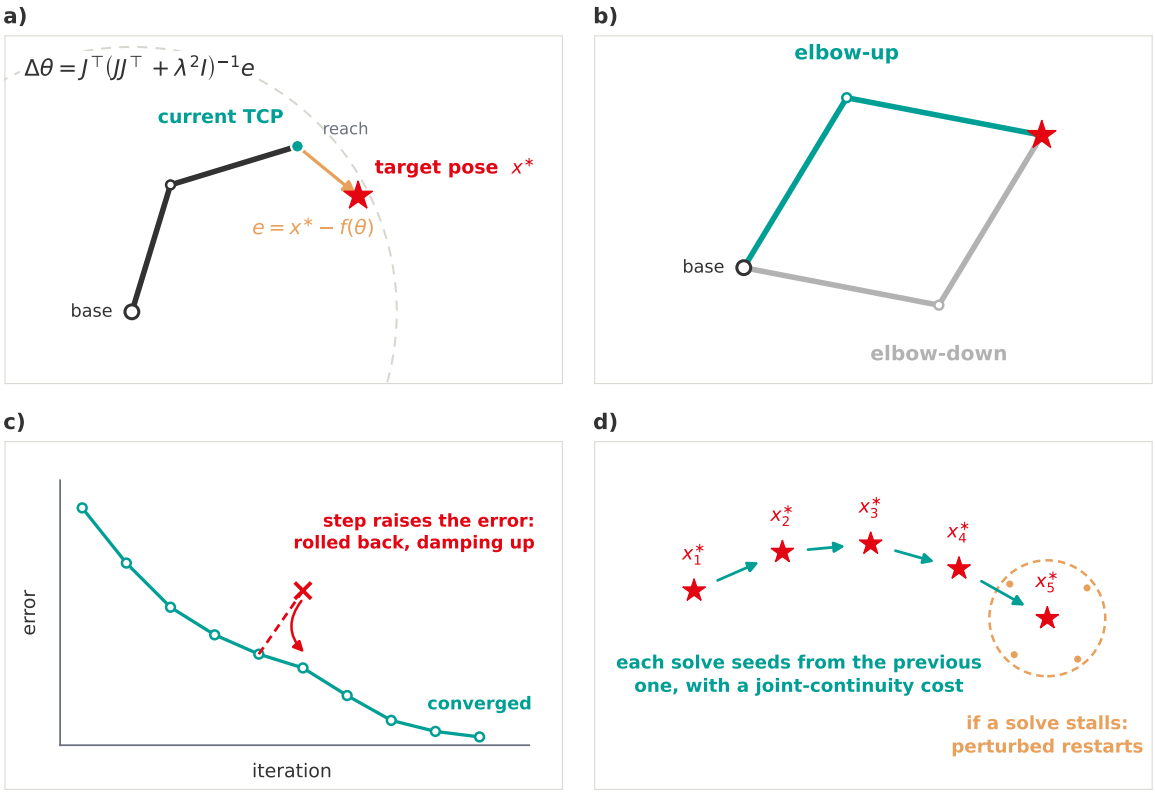


Figure 4.2: The four mechanisms around the inverse-kinematics solver (schematic). **(a)** the damped least-squares step along the full Jacobian, driving the pose error e to zero. **(b)** dual-branch seeding. **(c)** adaptive damping with backtracking. **(d)** warm starting along a gesture, with the randomized restarts used when a solve stalls.

4.3 Locomotion Catalog

Locomotion on the lattice is discrete, and the robot's complete gait vocabulary is a catalog of 61 actions in seven families. Figure 4.3 shows the families and follows two of the actions frame by frame in the simulator, while Table 4.1 recapitulates what each family covers. Although the catalog is large, it is built from a small set of independent concepts.

Vertical moves come in two sizes, *half* ($\pm\frac{1}{2}$ voxel, 37.5 mm) and *tall* (± 1 voxel, 75 mm), because the lattice is built from full and half voxels. Each climb additionally names its entry and exit *stance*, since on flat terrain both feet stand level while on a stair the feet are split by half a voxel, and the catalog covers all transitions. Every action exists in both leg *parities* (either gripper may lead), and a `switchLeg` action toggles the parity in place.

Each action is compiled to 22 inverse-kinematics keyframes on the real URDF, per entry stance,

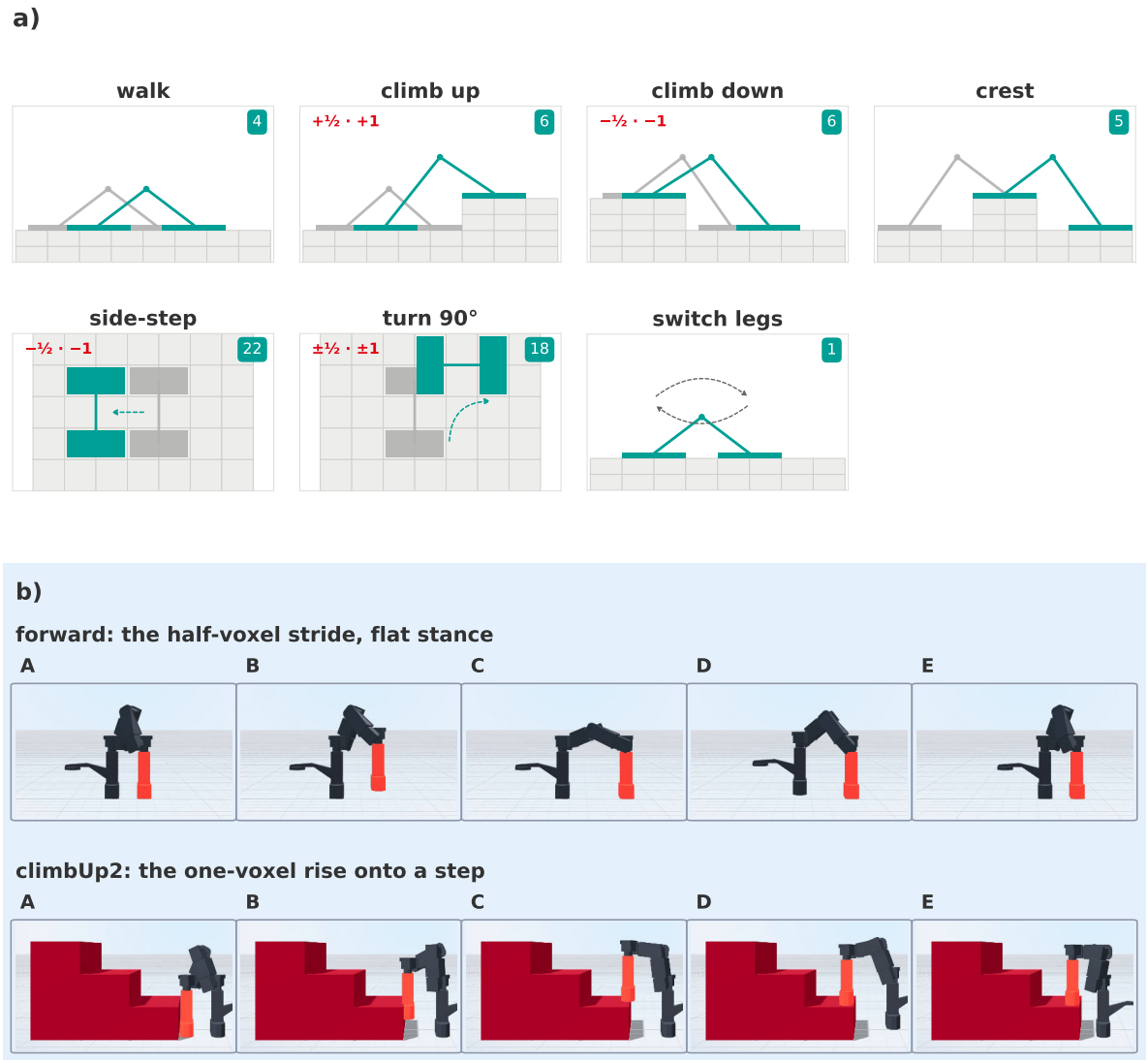


Figure 4.3: The locomotion catalog. **(a)** the seven families, in side view for the sagittal ones and top view for the lateral ones. Gray is the start stance and teal the end stance, while the red annotation gives each family's vertical vocabulary in voxels and the badge counts its actions. **(b)** two of them frame by frame in the simulator. Top strip, `forward`, the free foot arching over the planted foot to land half a voxel ahead (A–E). Bottom strip, `climbUp2`, the free foot swinging onto the step top and the body transferring up and over (B–E).

payload state, and parity, yielding a library of 1,286 motion clips. The core action set is verified exhaustively, with 660 of 660 runs (both parities, four facings, random map offsets) staying within 5 mm at every keyframe and showing no elbow-configuration jumps. Because each clip is validated once, offline, the planner can treat locomotion as a discrete search over a trusted action set.

Family	Actions	Height	What it does
Walk	4	level	`forwardHalf` and `goForward`, the half- and full-voxel strides on flat ground
Climb up	6	$+\frac{1}{2}$, $+1$	rises onto a step, over every entry and exit stance: flat to stair, stair to flat, stair to stair
Climb down	6	$-\frac{1}{2}$, -1	the same six transitions, descending
Crest	5	over a ridge	crosses a ridge or a dip in one motion instead of a climb-up followed by a climb-down
Side-step	22	level, $-\frac{1}{2}$, -1	travels one voxel laterally, or two in the *wide* variants, level or descending, left or right
Turn	18	level, $\pm\frac{1}{2}$, ±1	pivots 90° about the planted foot, left or right, on the level or while gaining or losing height
Switch legs	1	in place	swaps which gripper leads, so every other action exists in both parities

Table 4.1: The locomotion catalog by family. Action counts are the figure badges, and each action is further expanded into the compiled clip library (`robot/MOVEMENTS.md`).

4.4 Stance-Level Planning

Walking to a goal is a search over *stances* rather than positions, since a position the robot cannot stand in is not a place it can go. The state is the pair of foot positions on the voxel grid together with the heading and the moving leg, the actions are the catalog of the previous section, and the goal is a foot stripe to be reached with the right facing and parity.

The planner keeps the A* formulation of the previous pipeline [16] and updates it for the MILAbot v2 robot configuration and for the voxel set used here, so that it matches the machine that has to walk the path.

It is weighted A* with $f = g + 2.2\,h$, where the heuristic $h = \frac{1}{2}d_{xy} + \frac{1}{2}d_z$ (plus a constant 1.4 when the body faces the wrong axis) is an admissible lower bound on the remaining action cost. The bound holds because `goForward` covers two voxels for cost 1, climbs cover height at no less than 0.5 per voxel, and a turn costs at least 1.4. Since the estimate is smooth, it gives a gradient toward the goal everywhere, and since it never overestimates, the inflation factor is the only source of suboptimality. The search runs in a bounded region around start, goal, and terrain with a budget of 30,000 expanded nodes, raised to 50,000 for long walks while carrying, so it either returns a path or proves that none exists in the region.

Legality is enforced at every expansion rather than afterwards. Each 2-cell foot stripe must be unoccupied with clear headroom and at least one supported cell, step-height gaps must be within the action's limits, the swing path and the carried block must be collision-free, and foot parity must be preserved. A path returned by the planner is therefore executable by construction, which is the

same philosophy as the stability mask of Section 3.3, applied to locomotion. Figure 4.4 shows one such path being executed, with the robot walking to the stair foot and climbing twelve courses to the goal cell without a single replan.

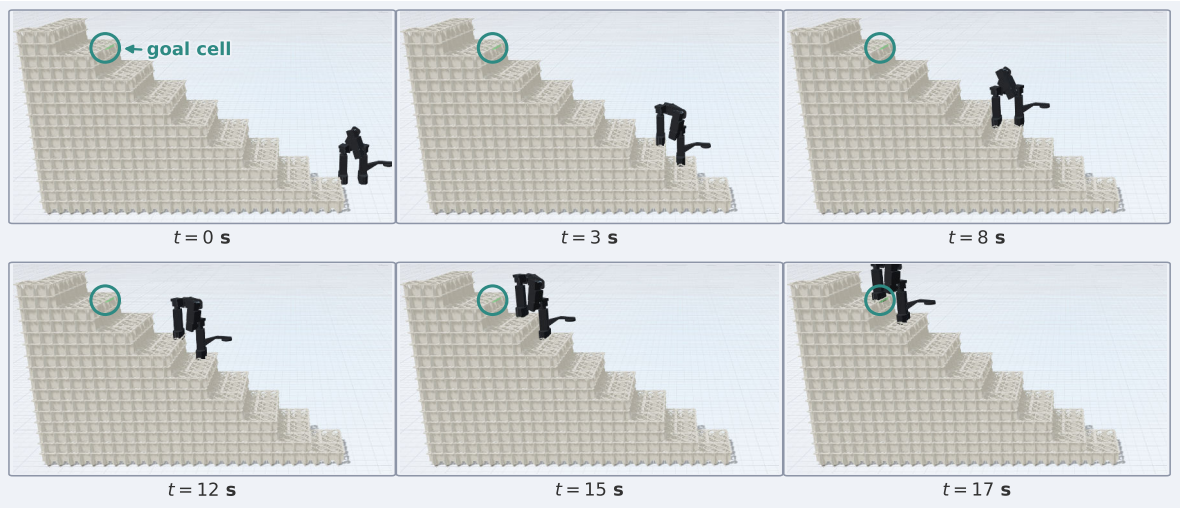


Figure 4.4: One planned path executed. Six frames of the traverse to the goal cell (ringed in teal), 17 s of simulated time. The camera is fixed and all six frames share one crop window, so progress can be read directly against the staircase.

4.5 Block Placements

A placement reuses everything above, being a *walk plus gesture* in which the robot walks to a stance near the drop cell using the planner of Section 4.4 and then executes a rotate-and-drop gesture around its planted foot. What the gesture can reach depends on the local terrain, so the simulator maintains 21 placement families in nine base styles, extended by lateral and forward offset variants.

The nine styles differ in how far the back foot travels and where the block ends up (Figure 4.5). *From behind* keeps the back foot planted and drops directly behind the stance. *From left* and *from right* swing it 180° around one side to place ahead. *Side* and *corner* styles swing 90° to place beside the stance, and *side-corner* styles spin 180° and step outward to reach further. Every family can drop its block at a height of zero, one, or two voxels relative to the stance, which is what lets one stance serve several courses. Figure 4.6 follows two of them end to end on different placements.

Every family is validated per placement before execution: IK feasibility for both legs at every keyframe of the gesture, foot-parity match with the arrival stance, and collision checks for the carried block through the full motion. When a family fails validation, the next one is tried.

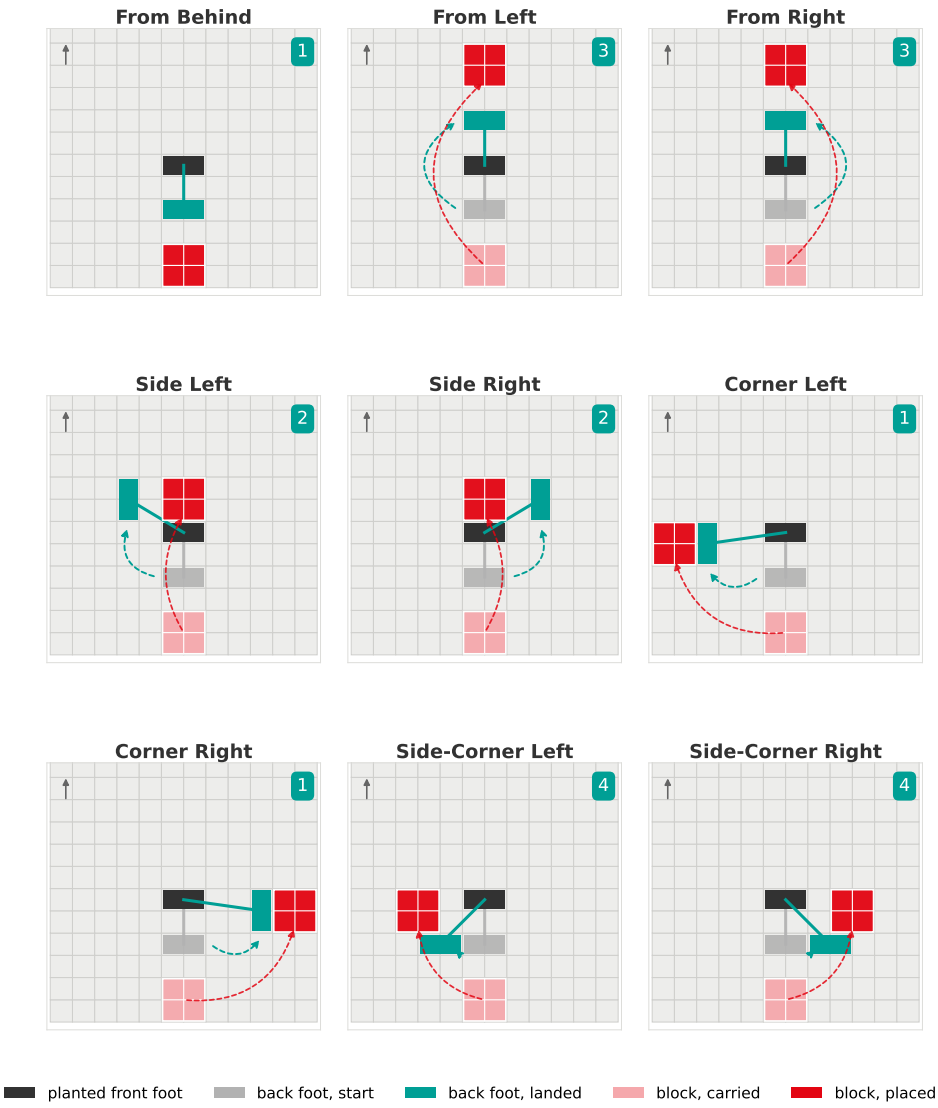


Figure 4.5: The nine placement gesture families, top view, with the robot facing up. The carried block (light red) rides on the robot, the back foot swings along the teal arc, and the block lands on the red cells. Badges count each family's offset variants, 21 in total.

4.5.1 Pickup and block orientation

In addition to positioning a block, every placement gesture also rotates it. The block is picked up from behind the robot by the payload arm and rides rigidly on the body, so its orientation is fixed at pickup. From that moment two things rotate it, namely the walk, where every 90° turn of the body turns the carried block with it, and the placement gesture itself, whose swing adds a fixed

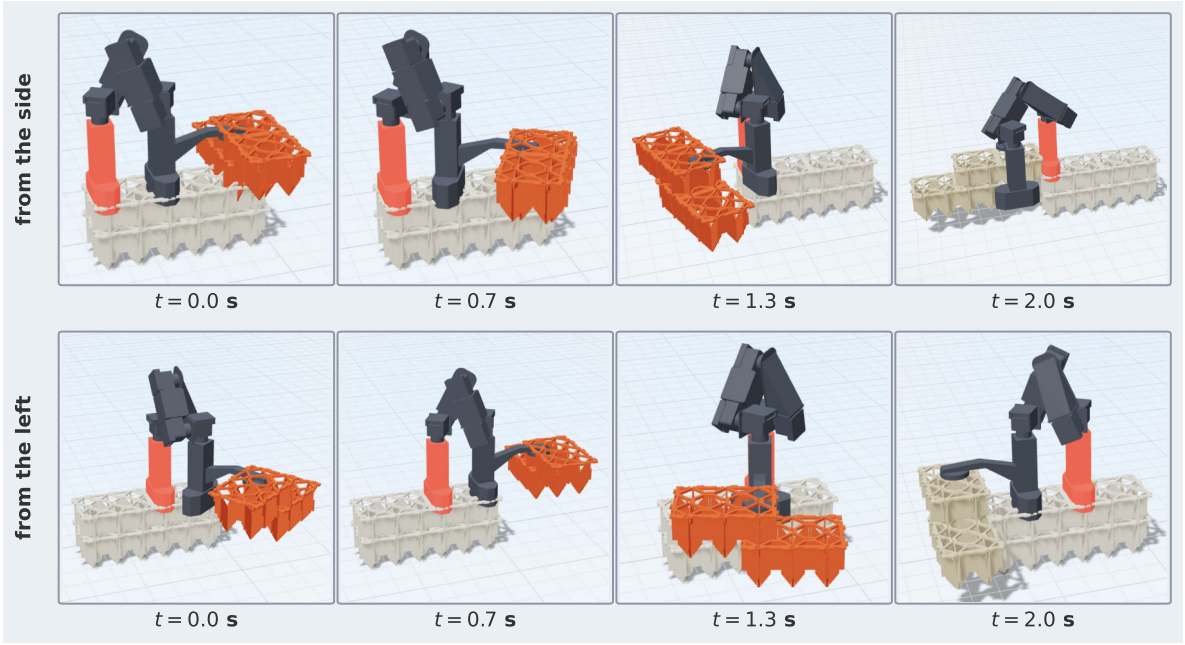


Figure 4.6: Two placements in simulation, each executed by a different gesture family, four frames apiece over a two-second gesture. Top, a *side* style, the back foot swinging 90° to drop the block beside the stance. Bottom, a *from-left* style, the foot swinging 180° around the left to drop ahead. The carried block is highlighted, and by the last frame it has joined the structure and taken its color. Frames are cropped to their content, since the viewpoint differs between captures.

quarter-turn per family (Figure 4.7). The placed orientation is therefore fully determined:

$$o_{\text{placed}} = o_{\text{carried}} + 90^\circ \cdot n_{\text{turns}} + \Delta_{\text{family}}, \tag{4.1}$$

where n_{turns} counts the quarter-turns walked between pickup and the placement stance and Δ_{family} is the family's quarter-turn offset. Blocks are not rotationally symmetric, since compound blocks such as 2×3 have a distinct footprint per orientation and the snap-fit interlock must meet the structure the way the pattern generator planned it. The build planner therefore inverts Equation 4.1, solving for the orientation the block must have at pickup given the target cell and the target orientation. An error in pickup orientation is unrecoverable downstream, so the feed must present every block in exactly its planned orientation.

4.6 Building Sequence

This layer orchestrates a full structure out of the single placements above. Its primary input is the stability-aware order of Section 3.3, and the RL planner's sequence is executed unchanged, so

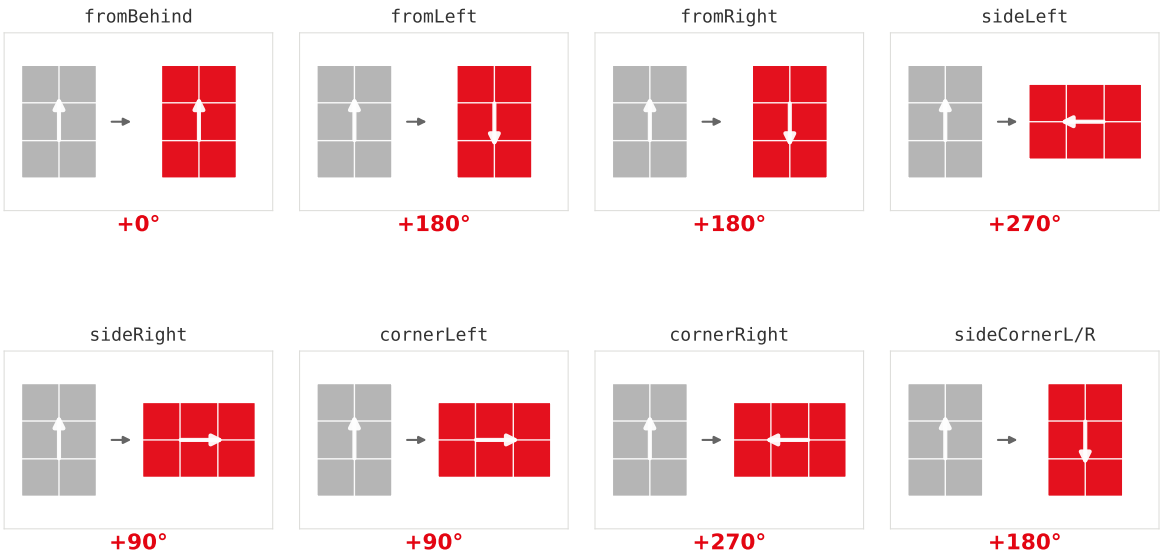


Figure 4.7: The quarter-turn each placement family adds between the carried orientation (gray) and the placed orientation (red), shown on a 2 × 3 block whose arrow marks its long axis. Swinging around to place ahead flips the block by 180°, while side placements rotate it by 90° or 270°.

the guarantees earned there carry through to execution. When no planned sequence is supplied, blocks are auto-ordered by height and then across the layer. In both cases the executor enforces two physical gates that the sequence alone does not capture.

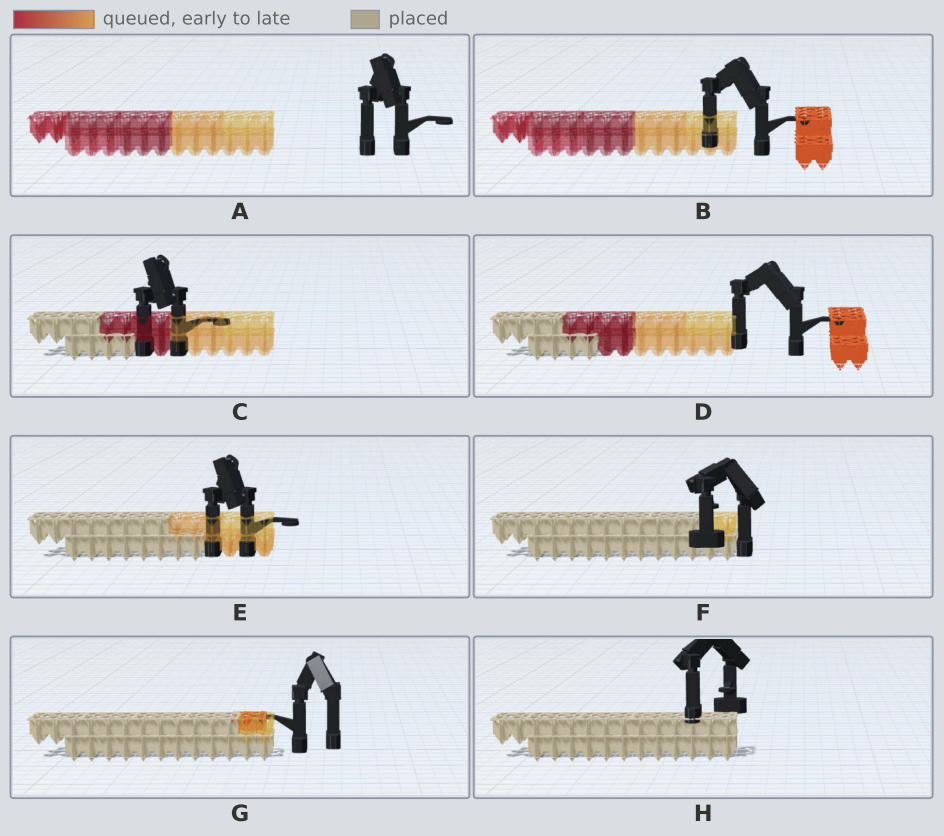
The first gate handles *staggered blocks*. Offset blocks interlock across columns, and the top course of a staggered block can overhang cells its own bottom course does not cover, so the floor under that overhang must be built by another block first. A naive position ordering can put the overhanging block ahead of its provider, and since each robot places its queue strictly in order, the build then stalls silently. The executor therefore runs a stable topological pass over each step's blocks, keeping the given order except where a stagger dependency forces a swap. Figure 4.8a shows the result on a staggered wall, which completes without a stalled placement.

The second gate is *collision-aware placement selection*, which decides which of the twenty-one families runs. Per block the loop is the same every time, planning the A* walk to a candidate stance and then trying placement families in a fixed preference order until one validates. The order follows how much airspace each approach disturbs. Side approaches come first because they move the foot least, then from-behind, which reaches only directly behind the stance, and finally the 180° swings, corners and side-corners, which reach furthest but sweep the most volume.

Each candidate is forward-simulated against the current terrain and the blocks already placed. The first family to survive is the one executed, so the choice is made by the terrain rather than by a

heuristic score. When a gesture that was free early in the build becomes blocked, the preference falls through to a family approaching from a side still open (Figure 4.8b). A block with no feasible family is reported while the sequence continues, so a failed run still leaves a usable partial build.

(a)



(b)

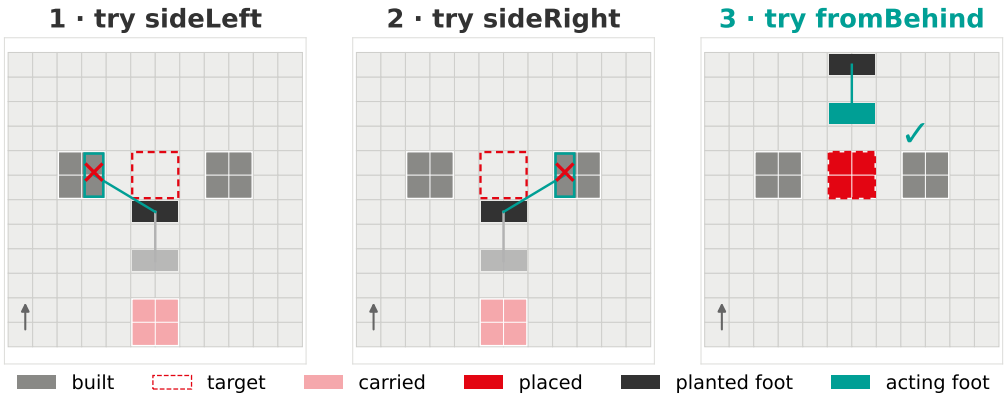


Figure 4.8: The two gates the executor applies to every block. **(a)** a staggered wall built in the planned order, with queued blocks ghosted and shaded early to late, placed blocks solid, and frames A–H sharing one camera. **(b)** collision-aware family selection for one target block (red dashed cells), top view, in the executor's try order. Both side swings land the foot in a built block, so the from-behind approach is the first to validate.

Chapter 5

Scalability and Hardware

The digital twin of the previous chapter simulates one robot faithfully, but it covers neither a whole site nor a physical machine. This chapter scales it to village level under real-robot constraints, treats the skin of a structure as geometry the planner has to respect, and then crosses the boundary to the fabricated MILAbot v2 and the live bridge that drives it.

5.1 Connected Pipeline and Village-Scale Builds

The previous chapters produced, separately, certified sequences and a robot able to execute block placements. This section connects the two and then scales the connection. A direct handoff binds the analyzer to the digital twin, and fleet orchestration coordinates multiple robots on one structure. The simulation then grows to *village scale*, meaning many structures, per-structure robot fleets and one merged scene. A final real-robot constraint, stepping back onto the just-placed brick, brings the simulation's behavior in line with the physical machine, at a cost we quantify.

5.1.1 Analyzer–Simulator Handoff

Once a structure has been analyzed, a single action, *Launch Simulation* in the analyzer, packages it into a payload containing the brick catalog, voxel cells, lattice size, robot count and support mode. The simulator decodes that payload and reconstructs the planned bricks on the shared 75 mm grid. If an RL-trained assembly order exists, it is passed through unchanged and executed as planned. Because both tools import one shared configuration (voxel size, grid conventions, block catalog), the structure the analyzer certified is by construction the structure the robots build.

The pipeline is thereby fully connected (Figure 5.1), with the evolution loop of Section 3.4 as the

feedback path for non-buildable targets. Every planned block maps to validated robot gestures, so a sequence can be tested end to end before any hardware moves.

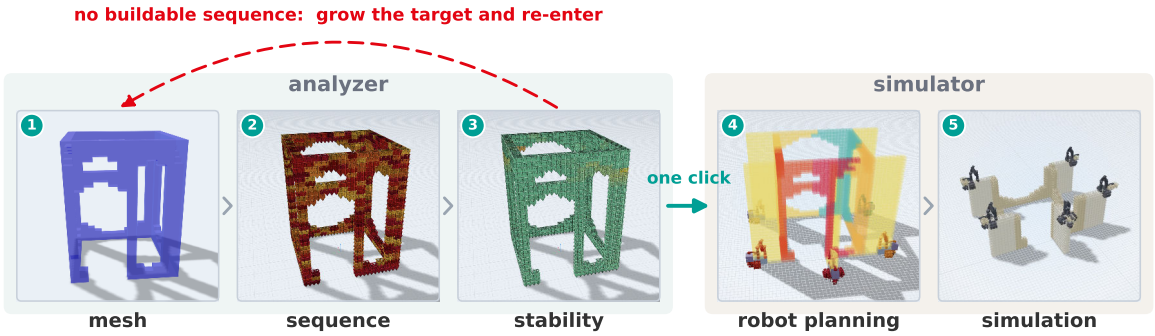


Figure 5.1: The connected pipeline, and where the two tools meet. The analyzer owns stages 1–3, voxelizing the mesh, planning a build order, and certifying that every intermediate state stands. The simulator owns stages 4–5, planning the robots' stances and gestures and executing the sequence in the twin. The single teal arrow is the whole handoff, one payload decoded on the shared 75 mm grid, and the dashed return is the evolution loop of Section 3.4, which re-enters at stage 1.

5.1.2 Multi-Robot Fleet Orchestration

While a single robot can walk a full structure brick by brick (Section 4.6), that approach does not scale. Fleet orchestration therefore partitions and coordinates the work by three rules, drawn schematically in Figure 5.3 and applied to a real target in Figure 5.2:

- **Work partitioning:** bricks are assigned to the nearest *stock stripe*, meaning the front/back pairs and left/right bands of material supply around the build site, so each robot draws from its own supply.
- **Course-by-course coordination:** within each course, climb supports are placed first, then structure bricks (Figure 5.3). A shared terrain store keeps every robot's world model current.
- **Gated waves:** placements are deferred until their support tier and work zone are ready, preventing robots from building themselves into dead ends.
- **Per-robot execution:** each robot walks to its pickup point, carries the brick, places it, and returns to the stock stripe. One robot plays at a time in the executor, and conflicts resolve through a defer-and-retry pass.

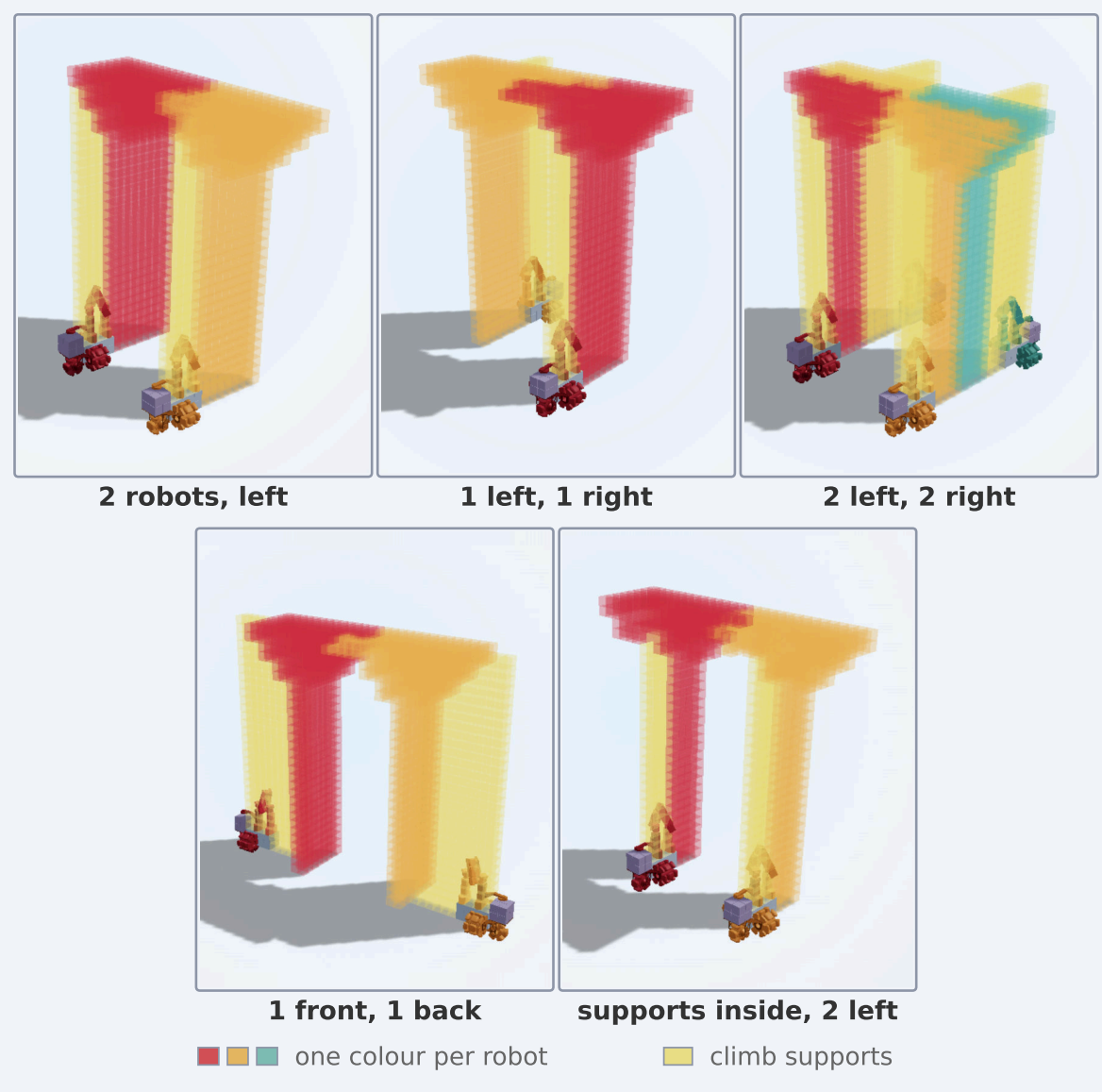


Figure 5.2: One target, five fleet configurations. Block colour is the robot the block is assigned to, while the pale yellow volume is the climb-support material the planner generates for that fleet. Adding a robot or moving one to the opposite face changes the plan without changing the target. The last panel is the case of Section 5.1.2, with the climb column dragged into the structure.

None of these rules is authored per structure. When the robots are moved, all three re-evaluate against the new stances.

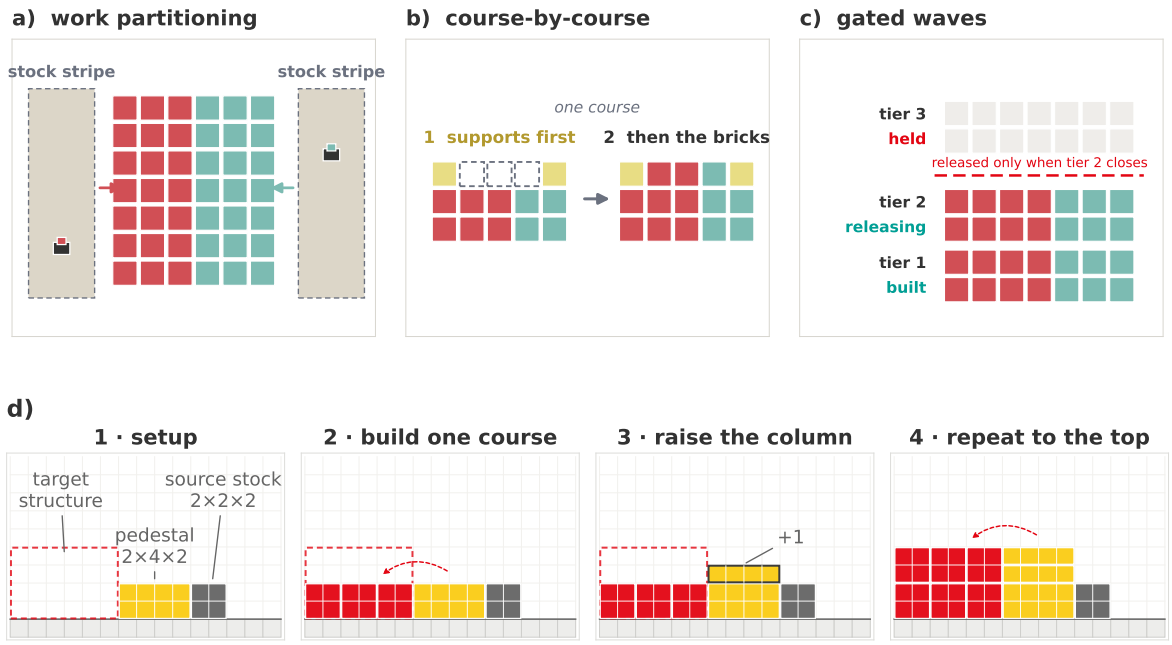


Figure 5.3: How a fleet is coordinated on one structure. **(a)** the work partition by stock stripe. **(b)** climb supports going down before the structure bricks that stand on them. **(c)** a tier released only once the tier below it is complete. **(d)** the climb-support system in four stages, a pre-placed $2 \times 4 \times 2$ pedestal flush against the structure face with source stock behind it (1), the course built from the column top (2), a $2 \times 4 \times 1$ climb plate raising the column by one voxel (3), and column and structure rising together (4).

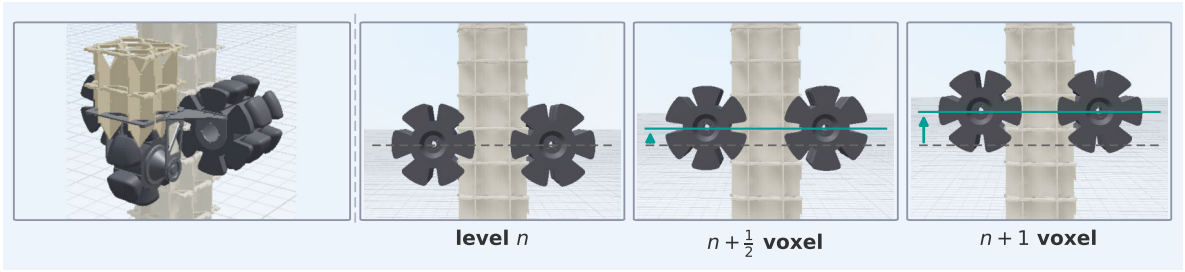


Figure 5.4: The elevator climbing its column. On the left, the machine straddles the climb column with a voxel loaded. The views on the right show one climb cycle at levels n, $n+\frac{1}{2}$ and $n+1$ voxels. The three frames are registered on the column rather than on the camera, with the gray dashed line marking the level the machine started from and the teal line the level it has reached.

Elevator

The climb column doubles as the track for the *elevator*: the machine that lifts voxels from ground stock to the working course (Figure 5.4, left). Two six-lobed drive wheels straddle the column and

engage the voxel faces directly, so the structure under construction is its own guideway and no separate mast has to be erected alongside the build. The track grows with every climb plate the robots place.

Because the lobes index on voxel features, the elevator advances in half-voxel increments, the same quantum as the robot's climb actions. In one cycle the drive wheels turn one lobe pitch from level n to $n+\frac{1}{2}$. A second turn completes the course at $n+1$, exactly the rise one climb plate adds to the column. Sharing the quantum keeps the schedule simple, since material delivery and robot access unlock a course at the same instant and the sequencer gates on a single height rather than reconciling two.

Supports inside structures

Climb columns need not stand beside the structure. Dragging a robot's climb column into a structure automatically reclassifies the overlapping bricks as support, and one climb plate per layer keeps the walking surface level with the next brick top. Material the structure needs anyway then serves as the robot's staircase.

Stepping back onto the placed block

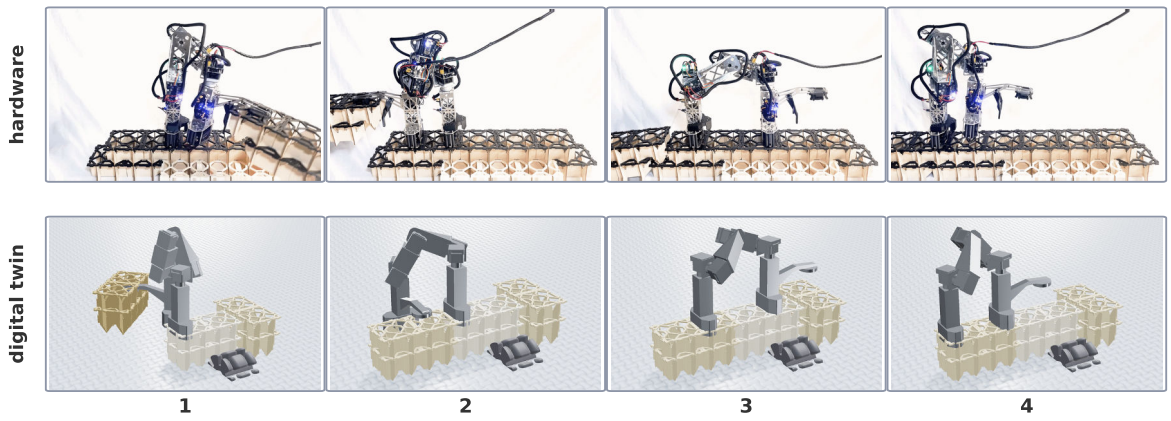


Figure 5.5: The step-on motion in four stages, executed by the physical MILAbot over a voxel bridge (top) and by the digital twin driving it (bottom). One foot stays on the completed course while the other plants on the newly placed brick and the body transfers over. The twin reproduces the stance sequence the planner certifies, meaning which cells carry a foot at what parity and height. It does not model cable drag, lattice compliance under load, or snap-fit settling, all of which the photographs show.

After placing a brick, the physical MILAbot steps *back onto it*, which the idealized simulation did

not capture, leaving a behavioral gap between plan and machine. *Real mode*, a per-robot toggle, closes that gap by generating step-on goals as 2 × 1 treads on the brick top after each placement, with the full path (place, step on, return) validated like any other (Figure 5.5). A robot for which no valid step-on path exists falls back to a normal return walk, so a build never fails because of the constraint itself. The constraint costs walking time and nothing else, as moves and final structures are identical with it on and off, and what that premium amounts to across five complete builds is measured in Section 6.4.

Four structures, end to end

Figure 5.6 shows four benchmark structures built end to end, each by the fleet its target was partitioned for. Build time spans two orders of magnitude, from 7 min 44 s for the simple bench to 2 h 20 min for the tiny house, and tracks block count and fleet size rather than structure height. These are simulated ideal-mode times on the twin's playback clock, measured with the step-on constraint off. The clock runs faster than the machine, so the four compare with each other rather than with a real build, and it is the spread between them that is the result here.

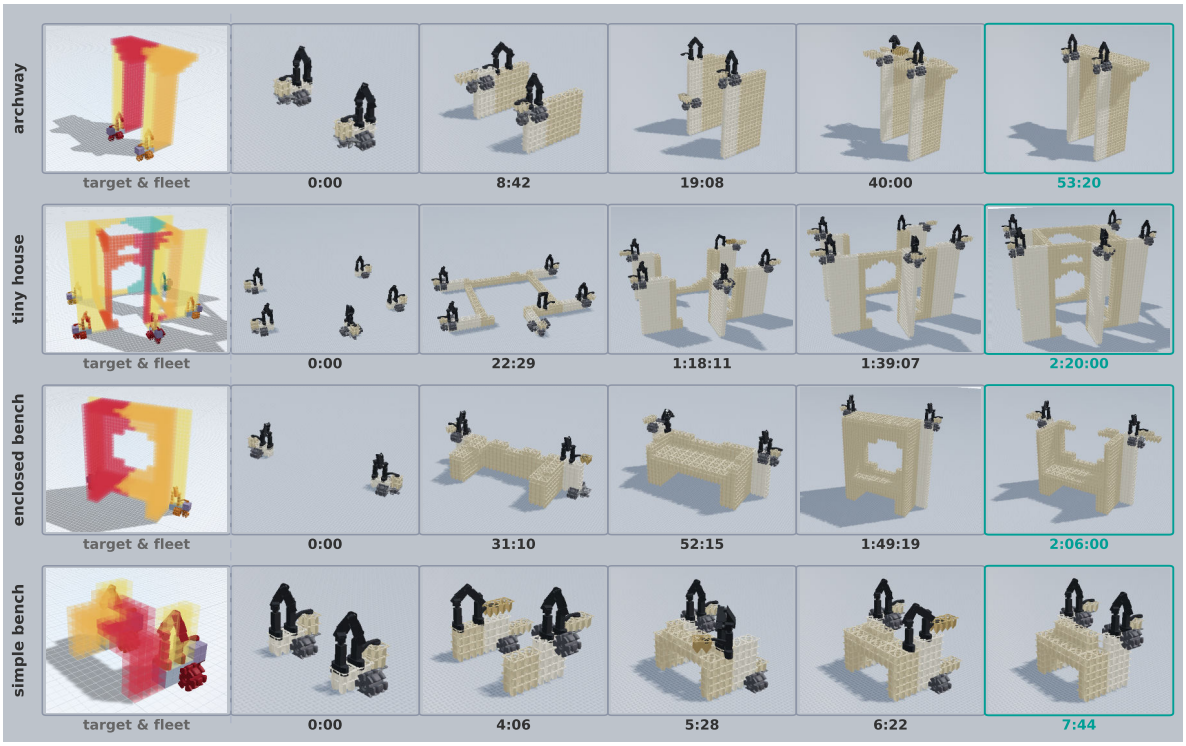


Figure 5.6: Four structures built end to end in ideal mode. Each row is one run, showing the coloured target with its robot assignment (left of the divider), then five snapshots of the build, from an empty site to the finished structure. Labels are elapsed simulated time, sampled from one continuous playback per run, so intermediate times are approximate and the final label of each row is the measured no-step-on build time.

5.1.3 Village-Scale Simulation

Since a site is rarely a single structure, the simulation has to carry several of them at once. The import schema (v2) accepts a `structures[]` list, each entry with its own blocks, name, and robot fleet, and all structures merge into a single layout. Whole structures can be duplicated, translated, and rotated anywhere on the lattice with drag gizmos, making site layout an interactive operation rather than a data-preparation step.

Consistency across the merged scene rests on two mechanisms. Robots are *namespaced* per structure (`structureId::robot`), so fleets never collide logically. Additionally, each robot's climb-support column follows its own structure's height, so builds of different heights proceed independently, and all structures build *in parallel* in one simulation.

The same machinery scales past benchmark objects. Figure 5.7 carries four house-scale structures (a pavilion, a dzong, a temple and a large house) through the same end-to-end runs. The targets are an order of magnitude larger and the fleets grow with them, yet the same course-by-course coordination, gated waves and climb-support generation run unmodified.

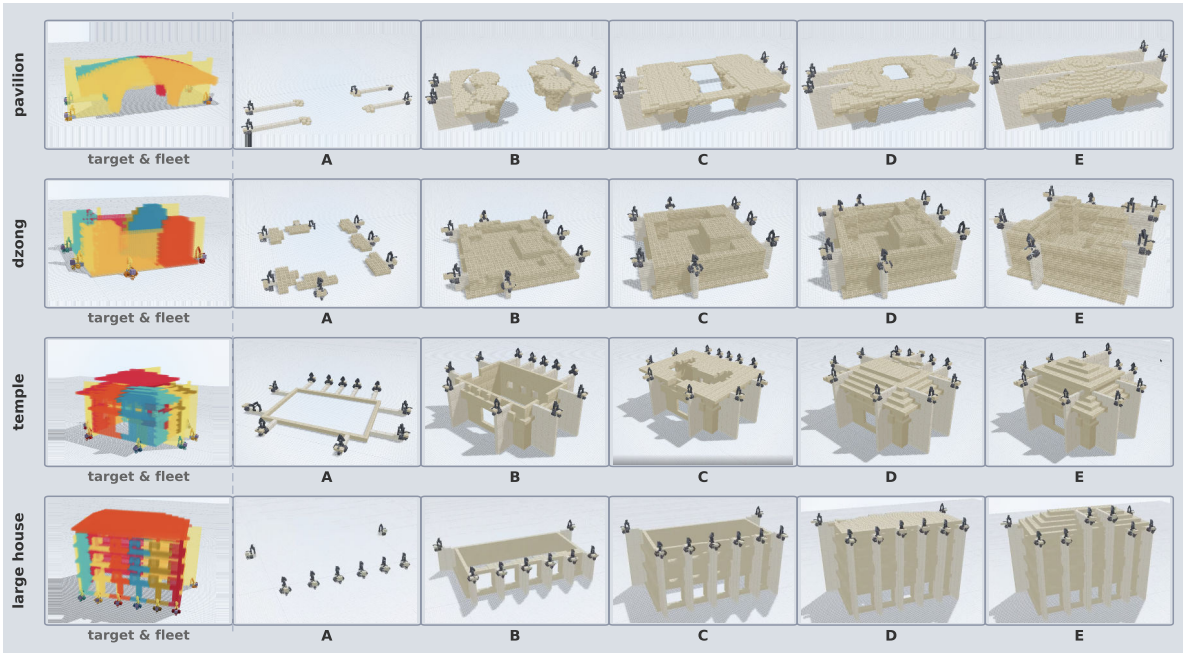


Figure 5.7: Four house-scale structures built end to end, in the same row layout as the benchmark builds and on the same unmodified planner and fleet coordinator.

5.2 Skins

A building needs more than structure. It has to enclose, shed weather, and, in the Bhutan context this project is aimed at, carry ornament that is central to the architecture rather than incidental to it. The voxel lattice is an unusual starting point for that, since its surface is already a regular grid of identical attachment features at one pitch. A surface system built on those features can therefore be placed by the machine that placed the structure.

5.2.1 Anchor Skins

The first exploration of that idea, carried out for this thesis's midterm (Figure 5.8), was a hybrid of rigid *anchor skins* and flexible patches. The anchor skin is a rigid panel that snaps onto a face of the lattice and covers a whole number of cells, 2×1 and 2×2 as the standard modules. It carries the pattern, since its face is flat and sized to tile into a composition, and it provides the discrete anchor points a flexible surface attaches to. The flexible half was to be thin bamboo veneer, light and reversible and locally sourceable, discretised into *curvagon* patches [39] rather than run as one sheet, so that approximating a smooth surface becomes a matter of fitting near-uniform patches and unrolling each to a flat cutting pattern.

This exploration was a long way from complete, amounting to a concept, a set of CAD studies, fabricated panels, and a fitting pipeline that did not yet preserve target geometry well. It did establish the part that turned out to matter, that a rigid module indexing into the lattice is the right interface and that the pattern belongs on that module.

5.2.2 Current Skin Design

The design was carried on from there by Vlasta Kubušová and Miana Smith, and the skin system as it now stands is theirs (Figure 5.9). Their panels retain the essential principle, but the pattern is cut into the panel rather than applied to it, so the ornament and the part are one object. The result is perforated rather than solid, filtering light and air instead of sealing, and weighing a fraction of a solid panel of the same footprint. Pattern is composed at more than one scale, from the motif on a single panel to the field across a column face to the profile running along the beam soffit and the crown. Because every panel still registers on the lattice, that composition follows from where the cells are.

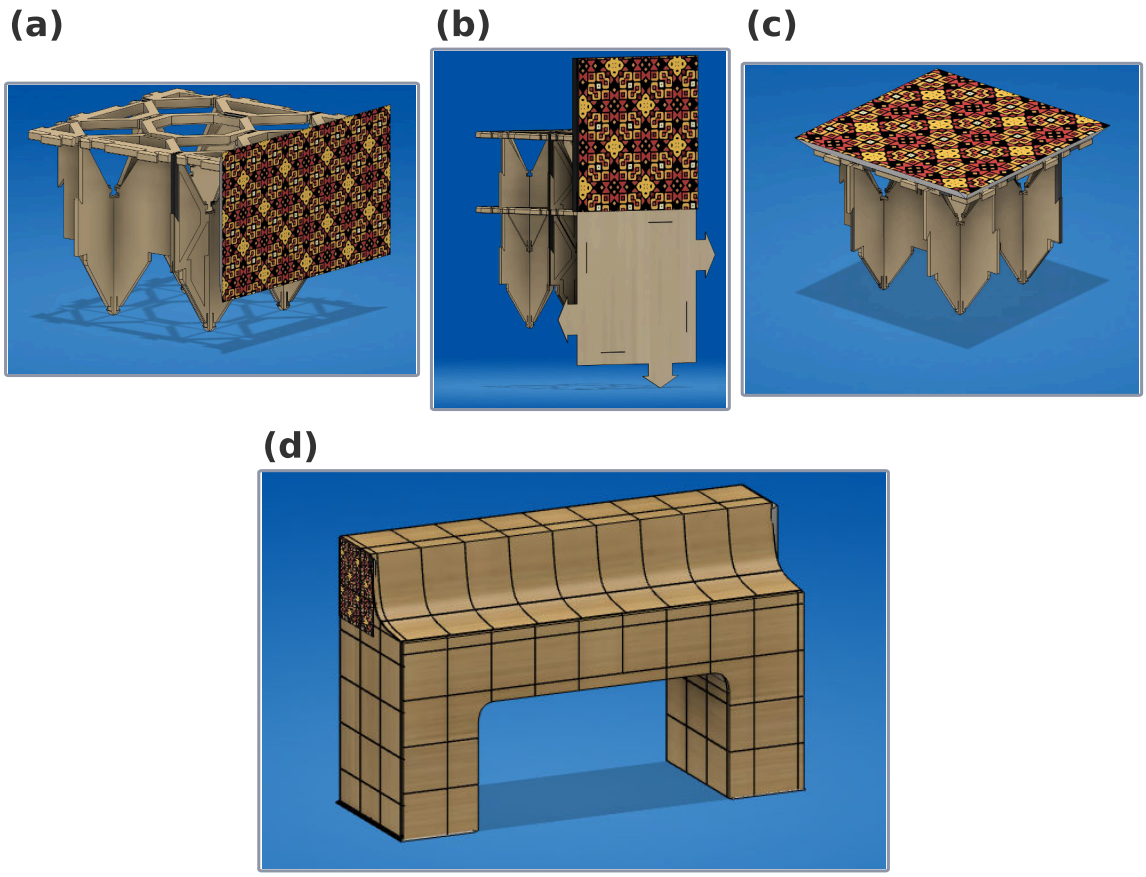


Figure 5.8: Early anchor-skin CAD studies. **(a)** a patterned rigid panel on the face of a 2 × 2 block. **(b)** the attachment geometry, with the panel pulled away from its anchors. **(c)** the same module used as a top skin rather than a wall skin. **(d)** a portal clad in panels, showing the tiling and the transition at the curved crown.

5.2.3 Skins in the Pipeline

Whichever panel design is used, the planner has to know the skin is there. It occupies volume on the faces the robot walks and reaches across, it adds mass to the blocks that carry it, and it has to be placed in an order that leaves it reachable. Carrying skins as first-class geometry is therefore the same argument the climb column makes in Section 5.1.2.

Figure 5.10 shows the tiny house built for Fab 26, a clad structure of exactly this kind, with the same geometry sitting in this thesis's simulator, tiled panel by panel on the voxel grid. The simulator reproduces the panel layout, the openings and the crown profile, which are what a plan has to respect. It does not reproduce the finish, the seams, or the way the pattern reads at grazing light,

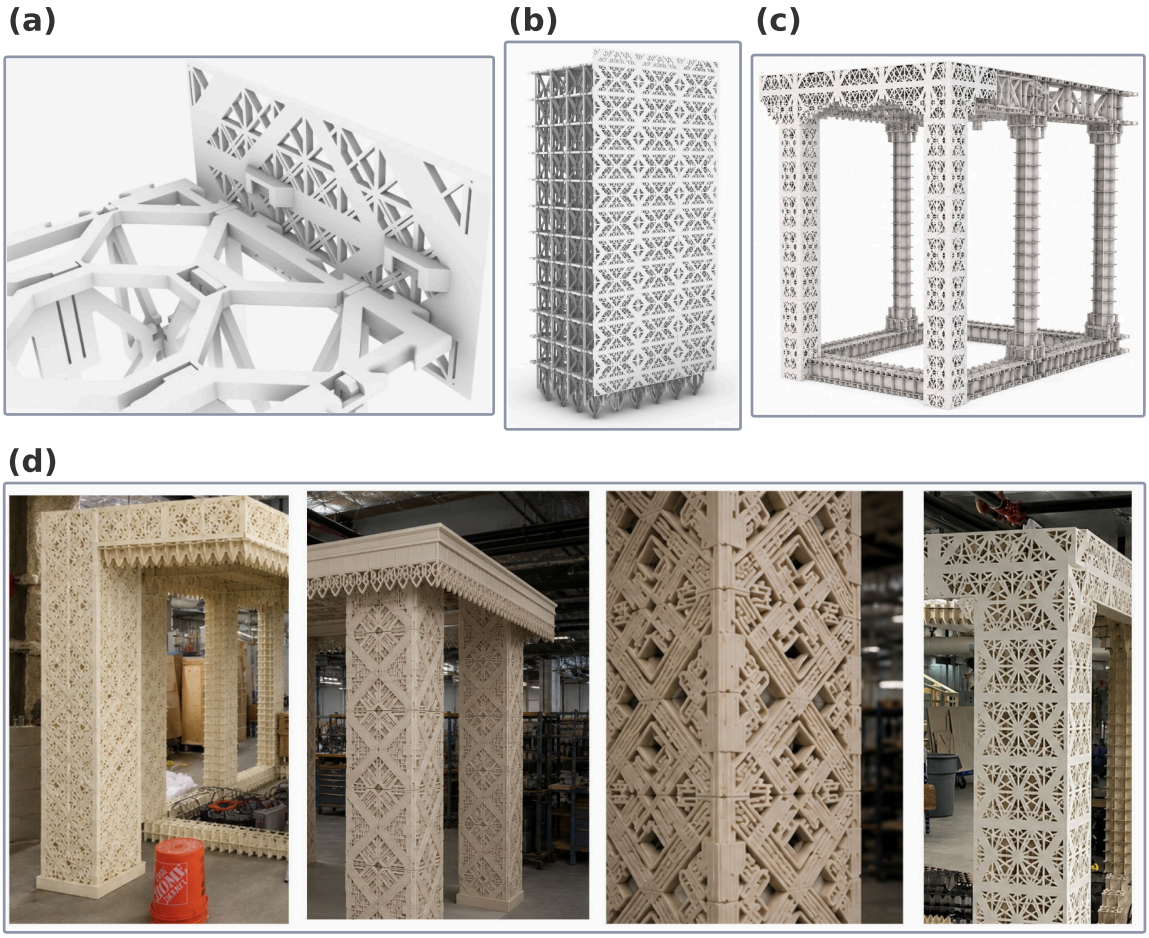


Figure 5.9: The skin system as developed by Vlasta Kubušová and Miana Smith. **(a)** a panel registering on the lattice. **(b)** panels tiled across a column face, and **(c)** a full portal. **(d)** the same system built, at four scales from the whole portal down to the joint between two panels.

none of which a planner needs and all of which are the reason the skin exists.

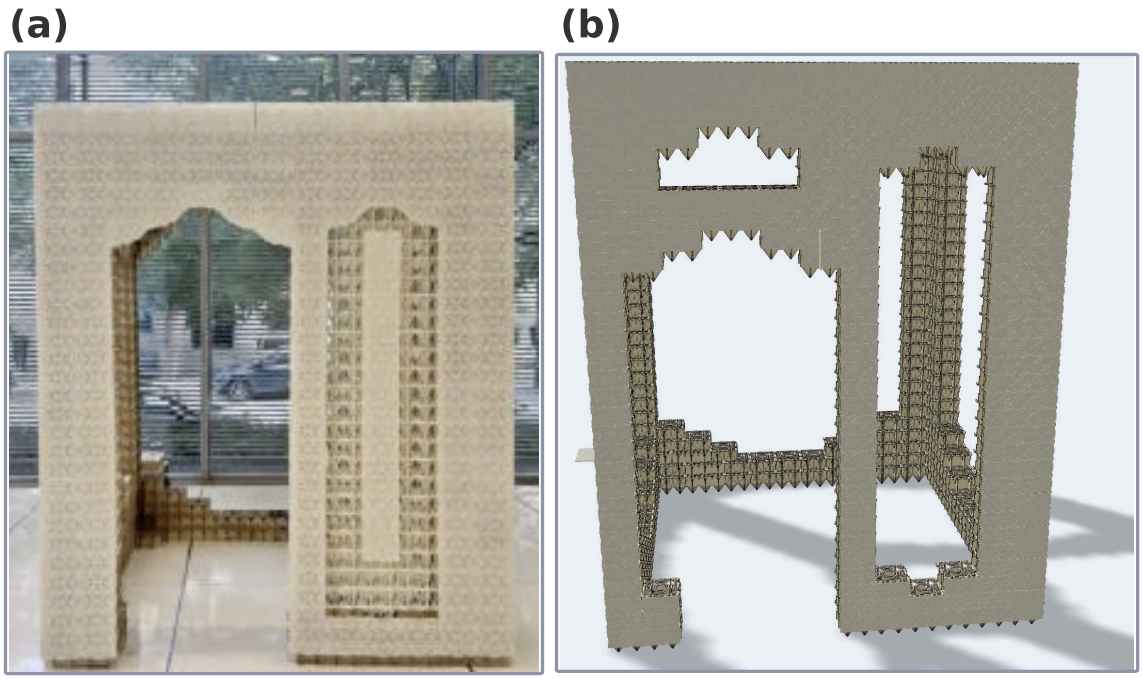


Figure 5.10: The Fab 26 tiny house, built and simulated. **(a)** the structure as built, clad in patterned panels. **(b)** the same geometry in this thesis's simulator.

5.3 Hardware Execution

5.3.1 Physical Robot

The MILAbot v2 used in this work was built from Miana Smith's v2 design [25] (Figure 2.2), with waterjet-cut structural parts, 3D-printed components, and an onboard ESP32. The voxel set is 3 mm plywood, with a revised lasercut design on 4.8 mm plywood as the fallback for a small demonstration set.

5.3.2 Hardware Link

The simulator drives the real robot live. A Python WebSocket bridge (`robot/hardware-bridge/`) listens on port 8765, and every simulator page carries a *Hardware* toggle that, when enabled, mirrors the selected robot's commands to the bridge as the build plays. The bridge relays them to the robot's ESP32 over WiFi as binary packets (command byte, I2C address, float32 payload). The link is deliberately non-blocking, in that a status indicator shows connectivity and the app retries

automatically. A dry-run mode prints every command instead of transmitting it, which is how every new sequence was checked before it was first sent to the robot.

The protocol is deliberately thin, in that the simulator sends the *name* of a motion rather than its trajectory (Figure 5.11). A `Macro:` message carries one whole gait macro identified by its simulator action id (`goForward`, `turnLeft`, `sideStepFlatRight`, `climbUpFlatStairTall`), and the robot expands it into its own tuned onboard motion. Three other messages complete the vocabulary. `Joints:` sends a direct five-angle pose in simulator-frame radians, which the bridge converts against the robot's calibration zero. `Cmd:` passes a raw string to the bridge's own parser for debugging. `Resume` releases a macro that has paused mid-sequence, so a pause can be cleared from the app rather than from the bridge terminal. Only two messages travel the other way, `PAUSED` and `RESUMED`, which is what lets the app show a Resume button at the right moment.

Naming motions rather than streaming them places the burden on the robot rather than on the plan. The mapping from simulator primitive to message lives in one table (`hardwareProtocol.ts`), so retuning a gait on the robot never invalidates a plan. Furthermore, the macro set itself is generated from the same validated clip library the simulator plans against (`gen_macros.py`) rather than being hand-maintained alongside it. The cost is that the robot must already own every motion a plan can contain, so an action with no macro cannot be mirrored, and the app warns per missing action instead of silently skipping it.

Execution is fire-and-forget, in that the simulator never waits for an acknowledgement and the bridge drains its command queue on a single worker thread in strict arrival order, so an unreachable robot cannot stall a build. However, nothing throttles the sender either, which is what bounds safe mirroring to 1×–4× playback.

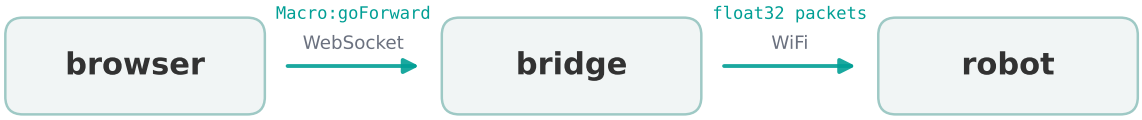


Figure 5.11: The hardware link. A planned move leaves the simulator as the *name* of a motion, crosses one WebSocket to the bridge, which turns it into joint poses for the robot's ESP32, which drives the servos over I2C.

The bridge was exercised on the motion that matters most for realism, the step-on of Section 5.1.2, with the physical machine and the twin driving it through the same four stages. Plans made without that constraint are not executable on the real machine.

5.3.3 Hardware Experiments

The structure built on hardware is a cantilever, chosen because it is the geometry the rest of this thesis exists for. A cantilever has nothing under its free end at any point during the build, so every intermediate state is carried by the connections alone. A geometry-driven sequencer has nothing to say about whether those states hold. The model of Section 3.1 returns a verdict on each of them, and the planner of Section 3.3 cannot emit a placement that breaks one.

Figure 5.12 follows one run of that build. The sequence was planned by the policy, certified block by block under the robots' own weight, and played in the twin, and the same sequence drove the physical machine over the bridge of Section 5.3.2. The robot walks out along the course it has placed, takes each block from the feed, and seats it beyond the last supported cell, so the overhang grows underneath the machine that is building it.

What the run demonstrates is the whole chain closing on one object, with the machine executing the stances the planner reasoned about on real voxels and without a replan. The two rows agree on which cells carry a foot, at what parity and at what height, and where each block lands. They do not agree on everything, since the photographs show cable drag, plywood compliance under load and snap-fit settling, none of which the twin models.

What the run does not demonstrate is scale or endurance. It is a single structure of a few courses, built once, under supervision, with the robot tethered. The questions a full-scale build raises, namely uptime over hours, drift accumulated over many placements, and whether two fronts meet where the plan says they will, are taken up in Chapter 7.

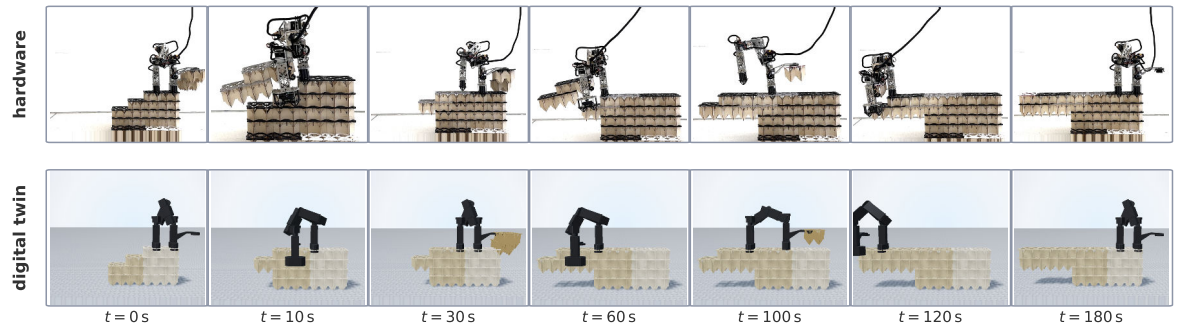


Figure 5.12: One cantilever, built twice. Top: the physical MILAbot placing the overhanging course. Bottom: the digital twin at the same points in the same sequence. Frames are sampled at the elapsed times shown and are content-cropped, so the two rows print at a comparable scale.

The step worth singling out is the one in Figure 5.13, where the overhang is fully erected and the robot is standing on it. This is the worst state the build passes through, since the deck is at its longest and the machine's own weight sits on the cantilevered course rather than on the grounded part of

the structure. The analyzer returns every block as balanced. The deck near the root runs warm on the ramp, where the snap-fits are closest to capacity, and nothing reaches the magenta end.

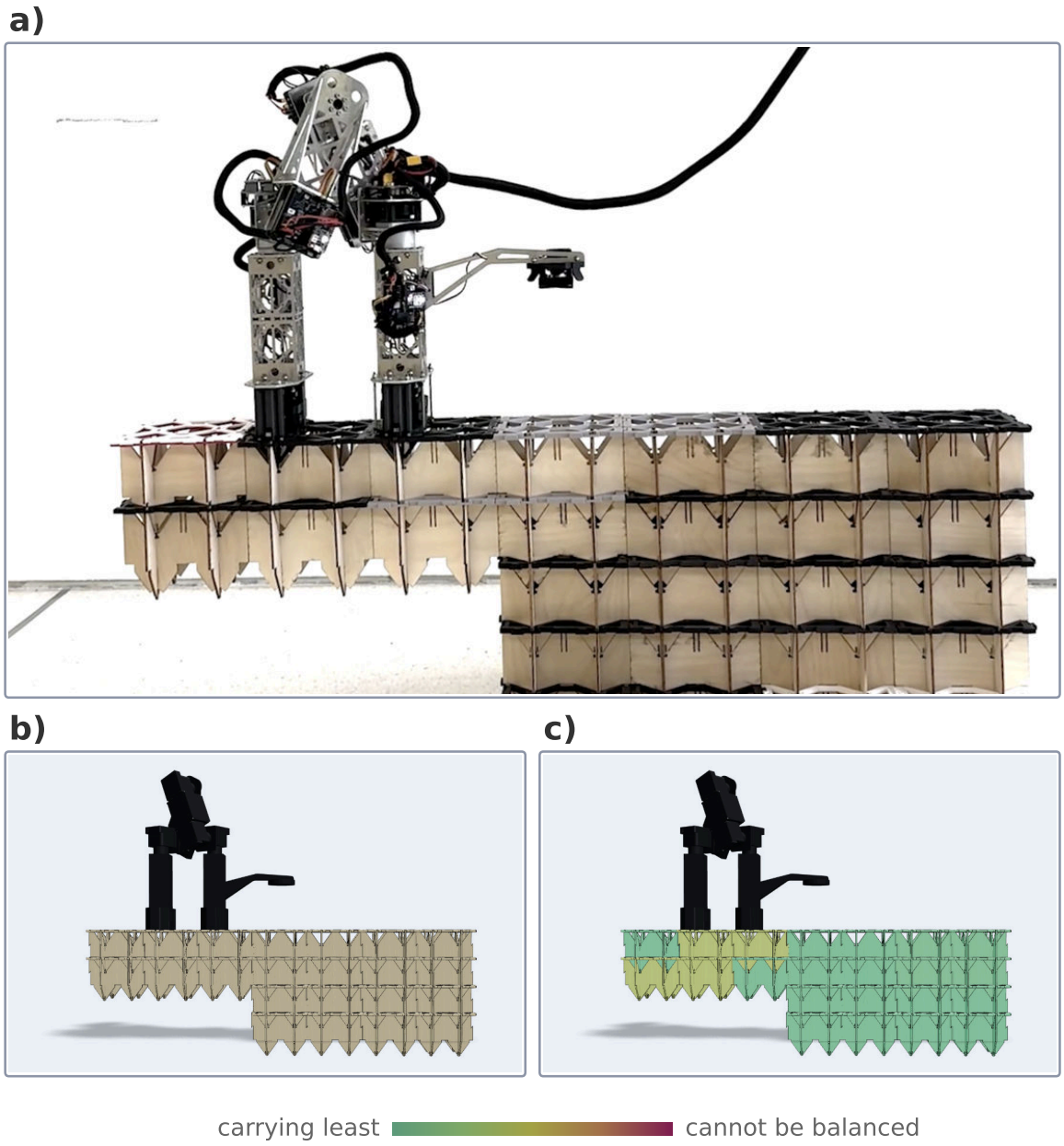


Figure 5.13: The stability verdict at the critical step of the hardware cantilever. **(a)** the state on hardware. **(b)** the same state in the digital twin, in wood colours. **(c)** the same state coloured by the per-block verdict of Section 3.1, on the ramp used throughout this thesis, where magenta is the colour reserved for a block that admits no admissible force assignment.

Chapter 6

Evaluation

Each of the three preceding chapters ended with a claim and a pointer to this one, and the corresponding results are presented in the order those claims were made.

Throughout the chapter, the benchmark suite is fixed and consists of eight structures spanning the failure modes of discrete construction, from the 96-voxel cantilever to the 2 992-voxel enclosed bench. The arche and the voxel cart join the suite wherever the question concerns scale or a fleet. Voxel counts are reported in Table D.3. Every measured number below was produced by the same code path the interface itself runs, on a laptop.

Two kinds of time are reported and should not be read alike. Training costs are real wall-clock on that laptop. Every build time, by contrast, is simulated, measured on the twin's playback clock, which runs faster than the machine. Simulated times are therefore comparable with each other rather than with a real build, and where one carries a result it is the ratio between runs that carries it rather than the number of minutes.

6.1 Patterns and Supports

Section 3.2 presented two ways of making an unstable structure stand, propping it up and laying it differently. Figure 6.2 compares both remedies on the same four targets, and the difference between them is an order of magnitude. Simple support columns add between 33 % and 432 % of the structure's own volume in sacrificial material that the robots must place and later remove, while the tree variant reaches the same stable verdict for 2 % to 61 %.

Patterning obtains the same stability far more cheaply, although neither hand-designed variant clears every target. Greedy layer-by-layer patterning leaves unstable voxels on all four, in the worst case 336 of 1568 (21.4 %) on the *covered shelter*, while the staggered-boss ILP clears three of them

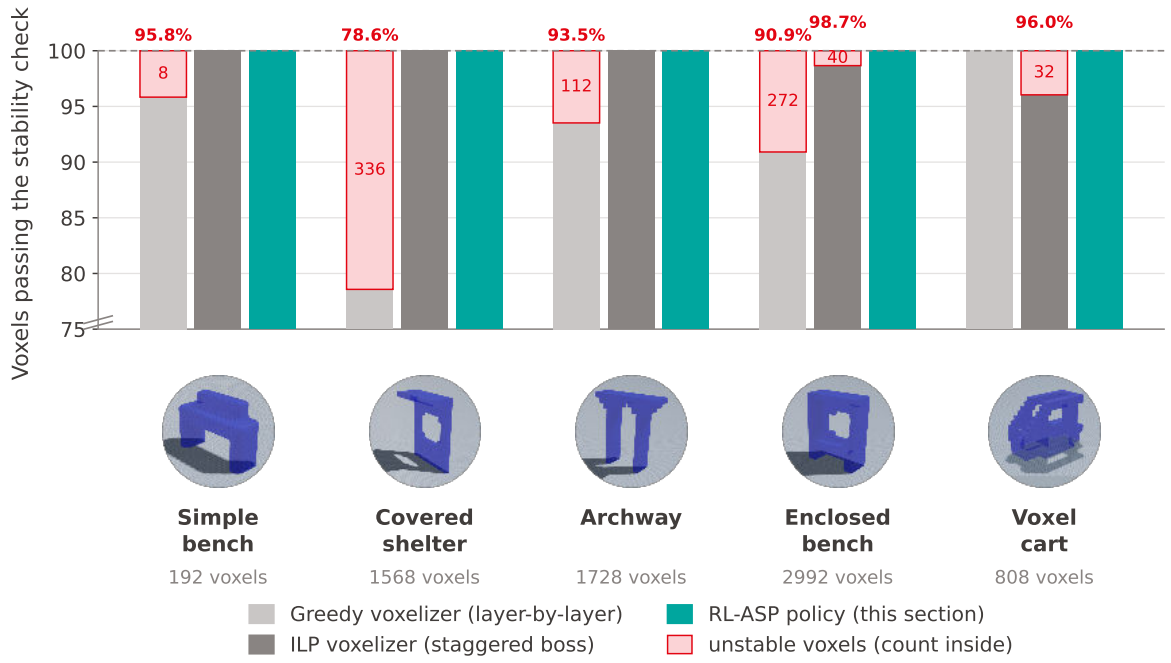


Figure 6.1: Stability verdicts for all three pattern strategies on the five benchmark targets. Each bar is the fraction of voxels that pass the analysis of Section 3.1. A bar reaching the dashed line has no unstable voxel, and a red cap is the unstable remainder, labelled with its count and its pass rate. The vertical axis is truncated at 75%.

outright and cuts the remaining failure to 40 of 2992 (1.3%) on the *enclosed bench*. Figure 6.1 shows both strategies across the suite, alongside the learned policy of the next section.

What the ILP cannot do is scale to architectural volumes, learn from anything, or say a word about the intermediate states. It is that third limitation that makes the next section necessary, since a pattern certified only in its finished form is not yet a plan.

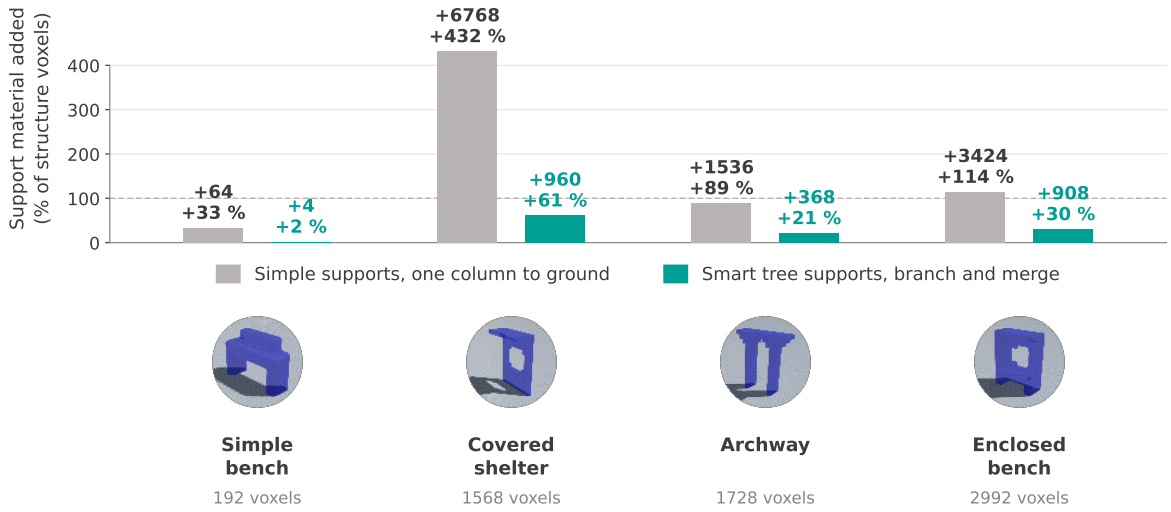


Figure 6.2: Support material added to reach a stable verdict on the same four targets, as a fraction of the structure's own voxel count. The bold number is the absolute count of support voxels, and the dashed line marks the point at which the supports cost as much as the structure itself. *Voxel cart* is absent because it needs no supports.

6.2 Sequence Planning

With the hand-designed strategies measured, three claims made for the planner remain to be checked: the space it searches is small, the plans it returns beat the hand-designed patterns, and producing one is cheap.

The first claim is read off the environment rather than estimated (Figure 6.3). On the enclosed bench, the largest target in the suite, the 90 112 conceivable placements of a 16^3 workspace, one for every cell against every block type in every orientation, fall to a median of twelve once the physics-aware mask is evaluated at each step. The two intermediate stages of that funnel account for most of the reduction, since restricting to the placements that fit inside the bounds leaves 49 860 and restricting further to those lying entirely on the target, which the placement tables precompute once, leaves 3 641.

Across the suite the median is five legal placements per step, with 90 % of build steps offering eighteen or fewer. The policy is therefore not searching a combinatorial space and being steered away from bad regions, but choosing among a handful of placements that are all sound by construction, which is also why a thousand environment steps are usually enough to settle a patch.

With the action space this small, the second claim concerns what the policy does with it. Over the full five-target suite, the policy leaves no unstable voxel on any target. The most informative

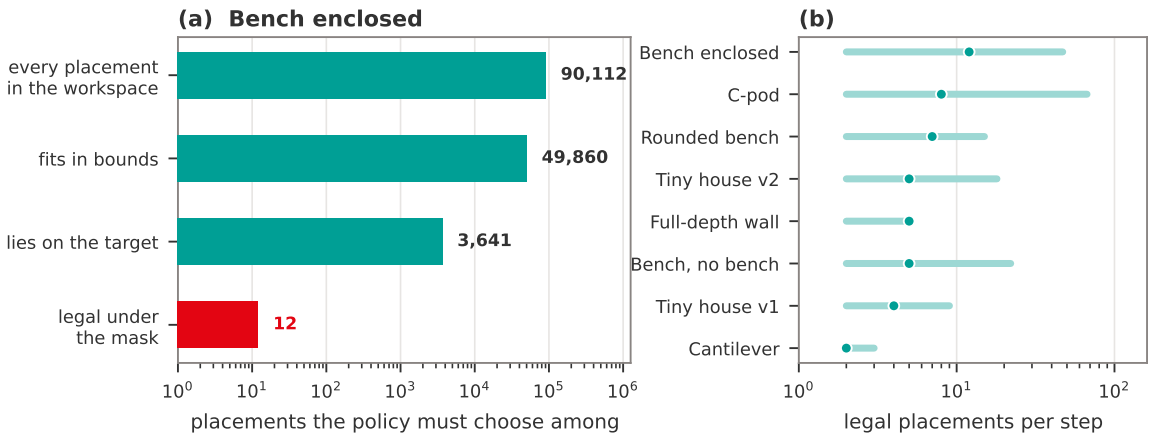


Figure 6.3: The action-space reduction, measured on the benchmark structures. **(a)** Successive stages of the funnel for a single target, the enclosed bench, on a log scale. The first two stages are the same for any target, while the last two are specific to this structure. **(b)** The branching factor the policy actually faces, for every structure in the suite, over 12 800 sampled build steps, where the dot is the median and the bar the 10–90 % range.

column of that comparison is *voxel cart*, the one target the ILP gets wrong and the greedy pattern gets right.

The staggering objective optimises seam alignment rather than the load path, so on a structure whose layer-by-layer pattern already carries load well, the ILP trades a good load path for a better-looking bond and loses 32 voxels. No choice of staggering weight fixes that, because the objective is not measuring the right thing. The policy has no such fixed objective to be wrong about, since the mask carries the physics.

6.2.1 Training Cost

The runs reported here were made with no imposed iteration limit, on a passively cooled laptop: Apple M2, 4 P + 4 E cores, 8 GB, CPU only, no GPU. The eight targets were voxelised at 75 mm by the analyzer's own voxeliser, so the grids are identical to the ones the interface trains on.

Every structure trains to a full-coverage build, and the whole suite costs 12.8 min end to end across 32 trained 16^3 patches (Figure 6.4). The smallest targets are done as a single window, the cantilever in 3.7 s, while the most expensive is the covered shelter at 278 s.

The two panels of that figure do not track each other, because the seconds spent per environment step are not constant. The archway spends 3.1k environment steps in 22 s and the C-pod 10.8k in 94 s, at about the same seconds per step. The covered shelter, by contrast, spends 278 s on 17.4k

steps, because its hard patch runs with a much larger legal-action set and a longer episode horizon.

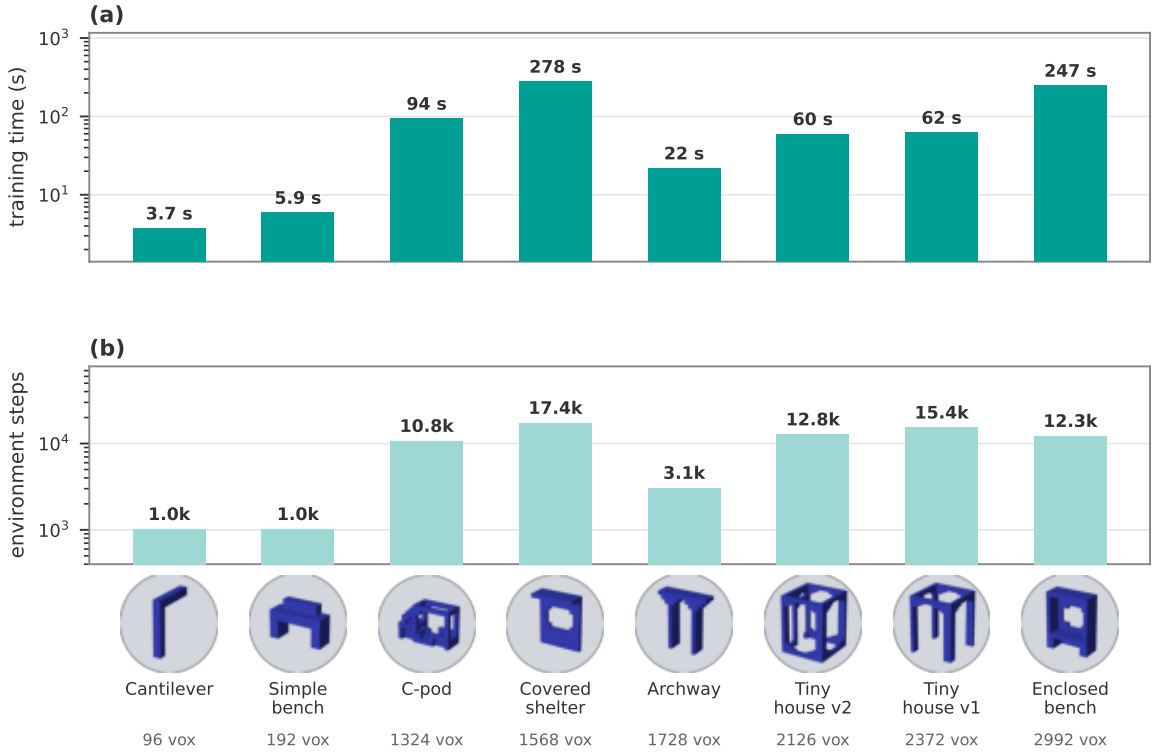


Figure 6.4: Training cost of the assembly-sequence policy on the eight benchmark structures, ordered by target size, where every run shown trains to a full-coverage build. **(a)** Wall-clock time, log scale. **(b)** Total PPO environment steps for the same runs, on the same axis, so that the two can be read against each other. Seconds per step differ by an order of magnitude across the suite.

Against target size the cost is close to linear, as a least-squares fit gives slope +1.08 ($r = +0.86$) for time and +0.84 ($r = +0.89$) for environment steps over this range (Figure 6.5). The scatter around that fit, however, matters as much as the trend, since seconds per voxel vary 14× across the suite, from 0.013 for the archway to 0.177 for the covered shelter. Those two structures are almost the same size, with 1728 voxels training in 22 s while 1568 takes 278 s, which is the pair circled in the figure. Size therefore sets the trend and the hardest patch sets the scatter.

The per-patch distribution shows where that scatter comes from (Figure 6.6). Across the 32 trained patches the median costs 1 024 environment steps and 5.2 s, and both distributions are strongly skewed. Half the patches are solved at the first evaluation, while the slowest single patch takes 16 384 steps and 270 s, on its own longer than any of the other seven whole structures.

The tiny house is larger than the covered shelter and decomposes into six times as many patches, yet trains in a fifth of the time, because the wall holds one stubborn patch while the house holds

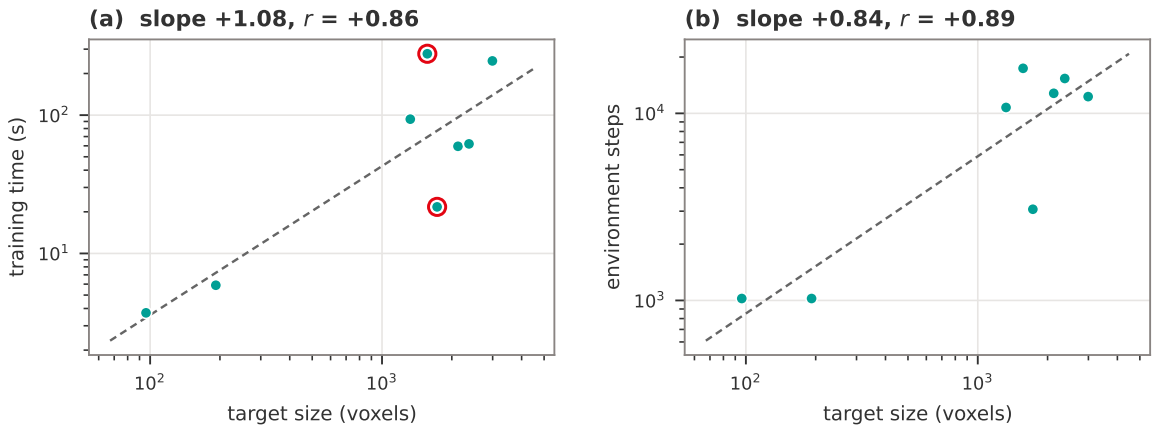


Figure 6.5: Training cost against target size, both axes logarithmic, one point per benchmark structure. **(a)** wall-clock seconds and **(b)** environment steps against voxel count, where the dashed line is a least-squares fit. The circled points in (a) are the two walls of near-identical size whose cost differs by an order of magnitude.

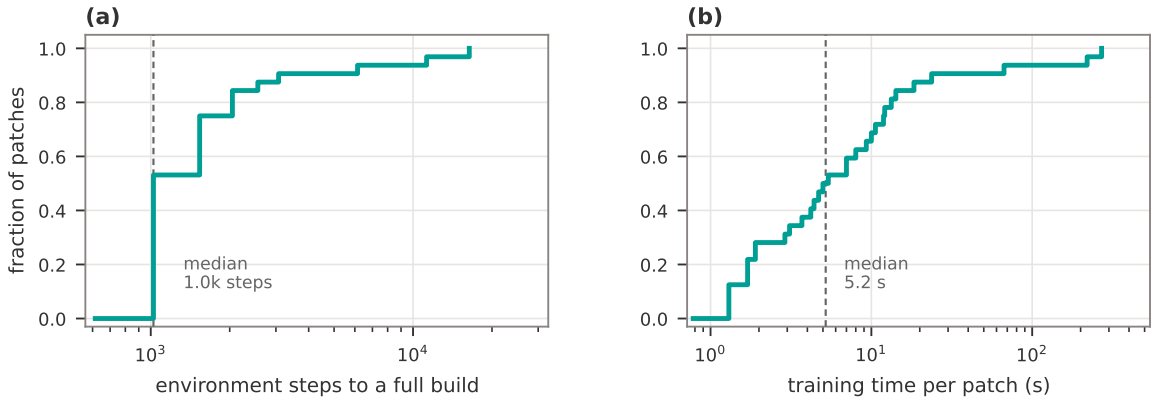


Figure 6.6: Cost of a single 16^3 patch, over every patch trained in the benchmark suite. **(a)** Environment steps to a full-coverage build and **(b)** the same patches in wall-clock seconds, both as cumulative distributions on a log axis. The tail is short but heavy.

twelve easy ones. Patch decomposition therefore provides more than parallel workspaces, since it isolates the hard part of a structure so that the rest stays cheap.

Three caveats belong with these numbers. All runs are single-seed per structure, and most structures are stable to a second or two between runs, the exception being the covered shelter, whose one borderline patch ranges 135–450 s over four runs while converging every time when uncapped. An earlier sweep with a five-minute cap recorded that structure as a failure twice in

four runs, which was an artifact of the cap rather than of the planner. Finally, targets whose layers have odd voxel area are excluded from this suite, because every block in the catalog covers an even number of cells within a layer, so such a target admits no exact tiling at all. Those targets are infeasible by construction rather than planner failures, and they are taken up in the next section.

6.3 Target Evolution

Growing the geometry rescues the targets the planner cannot build in every case tried, as thirteen of thirteen generalisation shapes, all seven arche patches and every hard monolith reach 100 % coverage of their original geometry. The cantilever used as the headline example goes from a 77 % dead-end to a complete build, at a cost of +128 support voxels and 246 s. Roughly half of the targets need no growth at all, with six of the thirteen shapes and two of the seven arche patches building as-is, which is what justifies the detect stage.

Table 6.1: Deterministic growth → warm-started build, measured on CPU. Coverage is of the original target.

Structure	Voxels (raw→grown)	Coverage	Bricks	Time
Cantilever, 3-overhang	42 → 52	100 %	10	5 s
Cantilever, deck	52 → 68	100 %	12	4 s
Bridge	240 → 264	100 %	43	9 s
Tower (5 × 7 × 10)	350 → 480	100 %	73	17 s

The targets that do grow gain between +44 and +192 voxels, about 15 % on the arche, and the amount follows geometry rather than size. On the arche, which exceeds one workspace and is evolved patch by patch (Figure 6.7), the patch carrying the arch itself grows twice as much as the solid piers. The detect-and-grow stage is negligible against the training that follows, at single-digit to low double-digit seconds on a CPU (Table 6.1), and the warm-started policies reach full evaluation success in 1 024 to 2 560 timesteps against a from-scratch budget of over 20 000.

One case is worth singling out, as the two-pier bridge went from needing +192 voxels to needing none on the warm start alone. Nothing about its geometry changed, and the planner simply got better at building it. Evolution is therefore a fallback for the limits of the planner as much as for the limits of the shape.

The known cost of this approach is a loss of minimality. The floor-fill pass makes the grower robust rather than minimal, so overhang-heavy shapes are offered more material than strictly necessary, which is exactly the surplus that Figure 3.16(b) shows the planner declining to use.

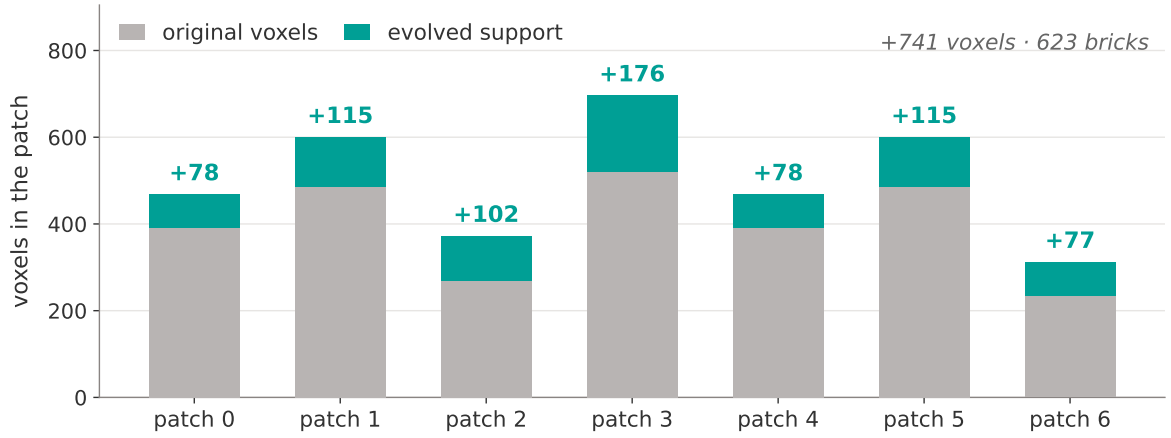


Figure 6.7: Patch-wise evolution of the arche, one 16^3 patch at a time. Each bar is a patch, where gray is the original voxel count and teal the support the grower adds before the patch becomes buildable. Every patch then builds to 100 % coverage.

6.4 Structure Builds and the Cost of Realism

A plan that plays in the twin still has to be executed by a robot that occupies the structure it is building, which is what the step-on constraint models.

The added realism costs walking time and nothing else. Across the five-build benchmark of Figure 6.8, moves and final structures are identical in both modes, so every difference between them is travel. The overhead is 26–37 % at two robots and 10 % at four robots on the voxel cart, which is consistent with a constraint that costs pure travel and with parallelisation absorbing part of it.

Both overheads are measured for sequential execution, with one robot moving at a time, and both modes run on the same clock, so the percentages are independent of it. Letting the fleet work simultaneously divides the build by the fleet size and more than repays the premium, taking the four-robot voxel cart from 1 h 11 min to 17 min 45 s. That parallel figure is an idealisation, since it divides the sequential build by the fleet size and carries none of the coordination overhead a real fleet would pay.

Village builds are evaluated on a multi-structure scene, reporting per-structure completion, wall-clock against the same structures built separately, and conflict statistics from the fleet coordinator.

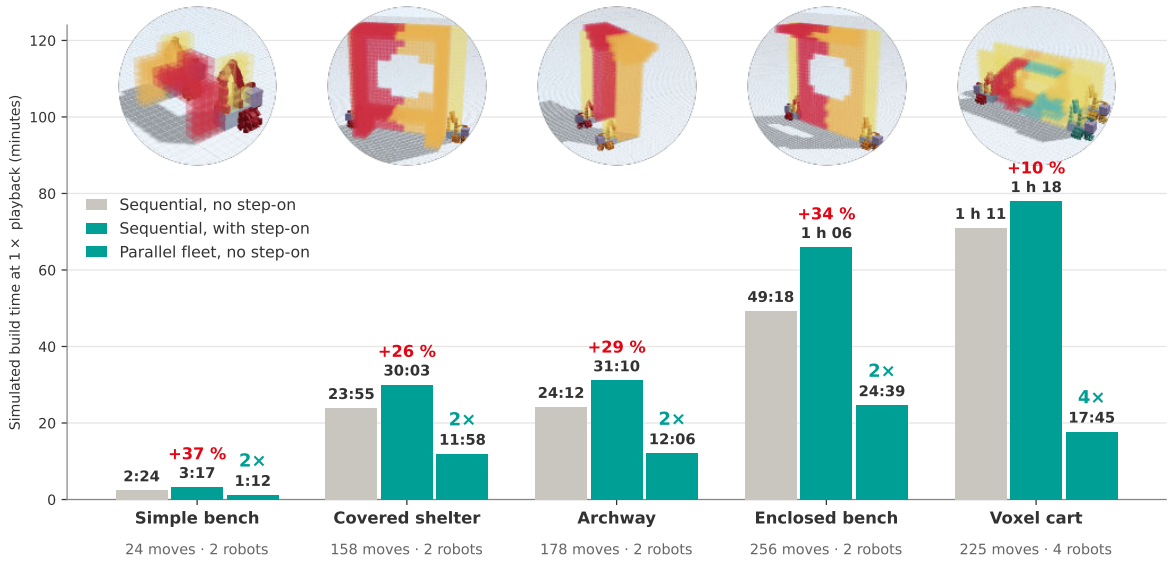


Figure 6.8: Simulated build time per benchmark structure in three execution modes: the ideal sequential build, the same build with the step-on constraint on, and the build with the fleet working in parallel. Stepping back on the platform costs 10–37 %, while running the fleet simultaneously divides the build by the fleet size, 2× for the two-robot builds and 4× for the four-robot voxel cart.

6.5 Build Cost and Throughput

Build cost is most easily read on a shape whose geometry raises no further questions, and three solid cubes were built end to end by one robot out of 2×2 blocks inside the 16^3 patch the planner works in (Figure 6.9). Simulated time runs from 2 min 22 s for the 4^3 cube through 19 min for the 8^3 to 4 h 09 min for the 16^3, so what the three runs establish is how the cost grows with edge length rather than any one of the three figures.

The placement count is simply the volume, since one move per 2×2 block means a cube of edge n needs $n^3/4$ placements. The predicted 16, 128 and 1 024 match the measured 18, 134 and 1 038 to within a few percent, the residual being the handful of extra moves the robot spends repositioning. Time, however, does not scale with the move count alone, as seconds per placement rise from 7.9 to 8.5 to 14.4 as the cube grows, because the walk between the stock stripe and the drop cell gets longer.

Beyond 16^3 the cube no longer fits one workspace, and the projection in panel (b) carries the same rule forward with one robot per 16^3 patch. That gives $(n/16)^3$ robots, so that a 32^3 cube takes 8 and a 128^3 cube takes 512. While the placement count keeps growing as n^3, the wall clock does not, because the patches are independent and each robot still builds one 16^3 patch, so the whole cube finishes in roughly the 4 h 09 min the single patch takes. This is the argument for a fleet of

small machines rather than one large machine, and it uses the same partition that the coordinator of Section 5.1.2 already applies.

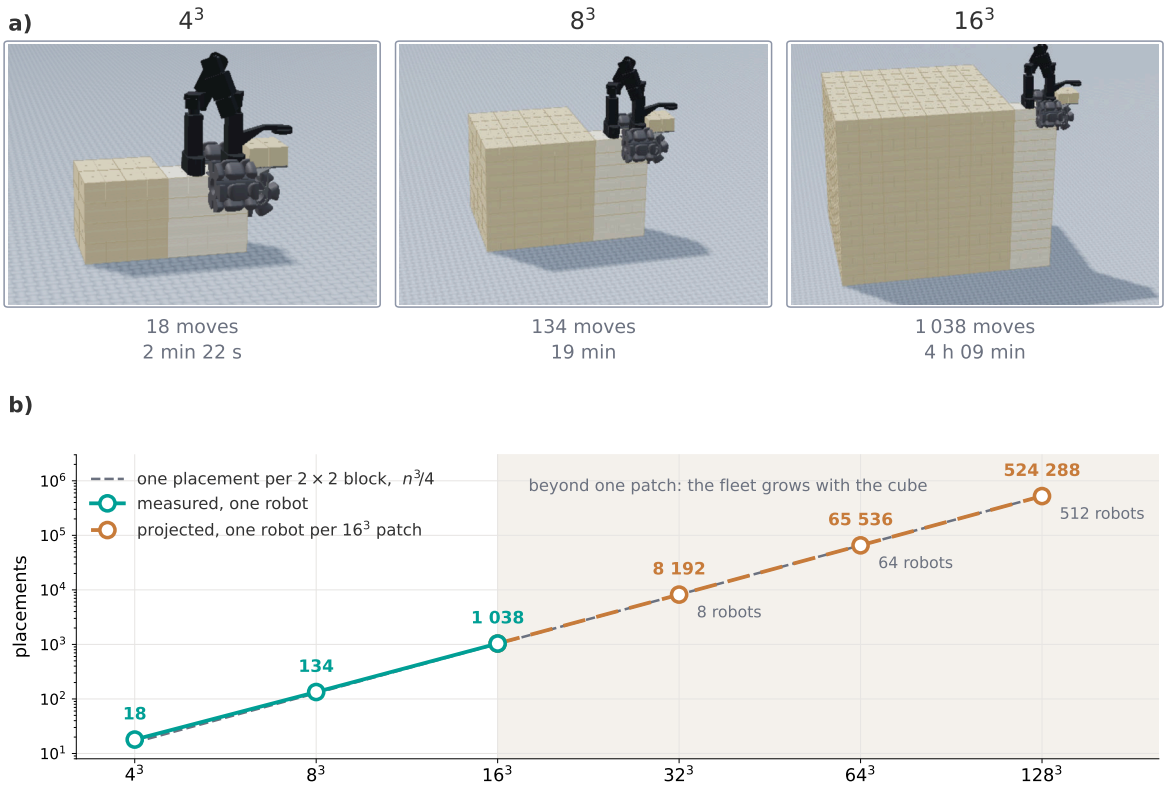


Figure 6.9: Build cost on solid cubes, in simulated time. **(a)** the three runs, one robot each, built out of 2×2 blocks. **(b)** placement count against cube edge, where the three measured points sit on the $n^3/4$ line. The projection beyond 16^3 (orange) is not a fit but the same rule carried forward, with one robot per 16^3 patch.

6.5.1 Throughput

In order to compare against machines that work nothing like a MILAbot, build time has to be converted into a volume rate. Richard's formulation [16] divides the volume a robot places by the time it takes to place it. Written per lattice cell rather than per carried block, which is what allows the two generations to be compared at all, it reads

$$Q_{MILAbot} = \frac{V_{\text{cell}}}{\mu \cdot t_{\text{move}}}, \tag{6.1}$$

where V_{cell} is the volume of one lattice cell, μ the average number of moves the robot spends per cell placed and t_{move} the average time per move. The per-cell form matters because v1 and v2 do not carry the same block, so a rate expressed per block would compare work per trip instead of work per unit of built volume.

For MILAbot v1 this rate is a model rather than a measurement. The robot carries three $4 \times 2 \times 2$ blocks per trip and spends 8.4 moves per block, and since a $4 \times 2 \times 2$ block covers sixteen cells that is $\mu_{\text{v1}} = 8.4/16 = 0.525$ moves per cell, at 9.5 s per move averaged over a hundred recorded movements:

$$Q_{\text{v1}} = \frac{65^3}{0.525 \cdot 9.5} = 5.51 \times 10^4 \text{ mm}^3/\text{s} = 3.30 \times 10^6 \text{ mm}^3/\text{min}. \tag{6.2}$$

This is Richard's own figure, since dividing the sixteen cells out of both sides of his expression leaves Equation (6.1) unchanged.

For v2 both quantities are timed on the physical machine rather than modelled, and both come out close to v1's. A move runs at 9.0 s against v1's 9.5, the small gain coming from the closed-loop actuator and from a foot that locomotes over a two-voxel-wide profile where v1 required a full 2×2 footprint for every step (Section 2.2). The robot spends the same 0.525 moves per cell. In their own terms the two machines are therefore about equally quick.

What differs is what a move is worth. The 75 mm cell carries 1.54× the volume of v1's 65 mm cell, so the same gait at the same cadence delivers half again as much structure per move:

$$Q_{\text{v2}} = \frac{75^3}{0.525 \cdot 9.0} = 8.93 \times 10^4 \text{ mm}^3/\text{s} = 5.36 \times 10^6 \text{ mm}^3/\text{min}, \tag{6.3}$$

or $1.62\,Q_{\text{v1}}$, of which 1.54× is the larger cell and 1.06× the quicker move. The improvement is in the block, not in the machine.

That the two generations spend the same moves per cell is the part worth stating plainly, because the two figures are not produced under the same conditions. Richard's 8.4 moves per block comes from a pipeline that performs no stability analysis at all, on the stated assumption that the structures it is given are already stable [16]. v2 meets the same move budget on sequences that are certified stable at every intermediate state and that carry the step-on constraint, whose cost was measured separately at 10–37 % of the wall clock (Figure 6.8). The comparison is therefore not a flattering one for v2: it holds a physics-aware planner to the move budget of a planner that ignored physics entirely, and the physics-aware planner meets it.

Taken with the unit prices, v2 is also the better buy, at 5.36×10^6 mm^3/min for 1700 USD against 3.30×10^6 for 1250, or 1.19× the throughput per dollar. The extra 450 USD therefore returns more volume rate than it costs, before counting the closed-loop actuator, the reach and the payload

gripper that make an autonomously planned, stability-certified build executable at all, none of which appears in a volume rate.

6.5.2 Comparison with Other Autonomous Construction Systems

The same rate for a continuous printer is its nozzle speed times the cross-section it lays down, and for a brick-laying machine its brick rate times the brick volume [16]:

$$Q_{\text{printer}} = v_{\text{nozzle}} \cdot \left(w_{\text{layer}} \cdot h_{\text{layer}} \right), \tag{6.4}$$

$$Q_{\text{brick}} = r_{\text{brick}} \cdot V_{\text{brick}}. \tag{6.5}$$

Applied to the three systems surveyed in Section 1, and with every rate expressed per minute, Equation (6.4) gives $12\,000 \cdot 30 \cdot 30 = 1.08 \times 10^7$ mm^3/min for the arm-mounted MaxiPrinter, whose nozzle runs at 200 mm/s over a 30×30 mm cross-section, and $15\,000 \cdot 40 \cdot 300 = 1.80 \times 10^8$ mm^3/min for the COBOD gantry at 250 mm/s over 40×300 mm. Equation (6.5) gives $6 \cdot 7.2 \times 10^7 = 4.32 \times 10^8$ mm^3/min for the Hadrian X, whose 360 bricks per hour of $600 \times 400 \times 300$ mm are six bricks per minute.

On raw throughput a single MILAbot loses to all three systems by one to two orders of magnitude, which has never been the argument for it. The argument is throughput per dollar, together with the fact that both scale by adding units. A fleet of N machines costs Nc and delivers NQ, so matching a system of throughput Q_{sys} takes

$$N = \left\lceil \frac{Q_{\text{sys}}}{Q_{\text{v2}}} \right\rceil \quad \text{units, at a cost ratio} \quad \frac{p_{\text{sys}}}{Nc}, \tag{6.6}$$

with $c = 1700$ USD per MILAbot v2 unit (1200 for the robot and 500 for the elevator and feed). Figure 6.10 plots Equations (6.1) to (6.6) on the axes Richard uses. Three MILAbots match the arm-based printer for roughly 78× less money, 34 match the COBOD gantry at 15× less, and 81 match the Hadrian X at 43× less, so that buying throughput by adding small machines stays cheaper than buying it in one large machine by one to nearly two orders of magnitude. The setup cost differs in the same direction, since a MILAbot fleet needs the robots positioned and their feed stations placed, whereas a gantry of comparable output has to be erected on site first.

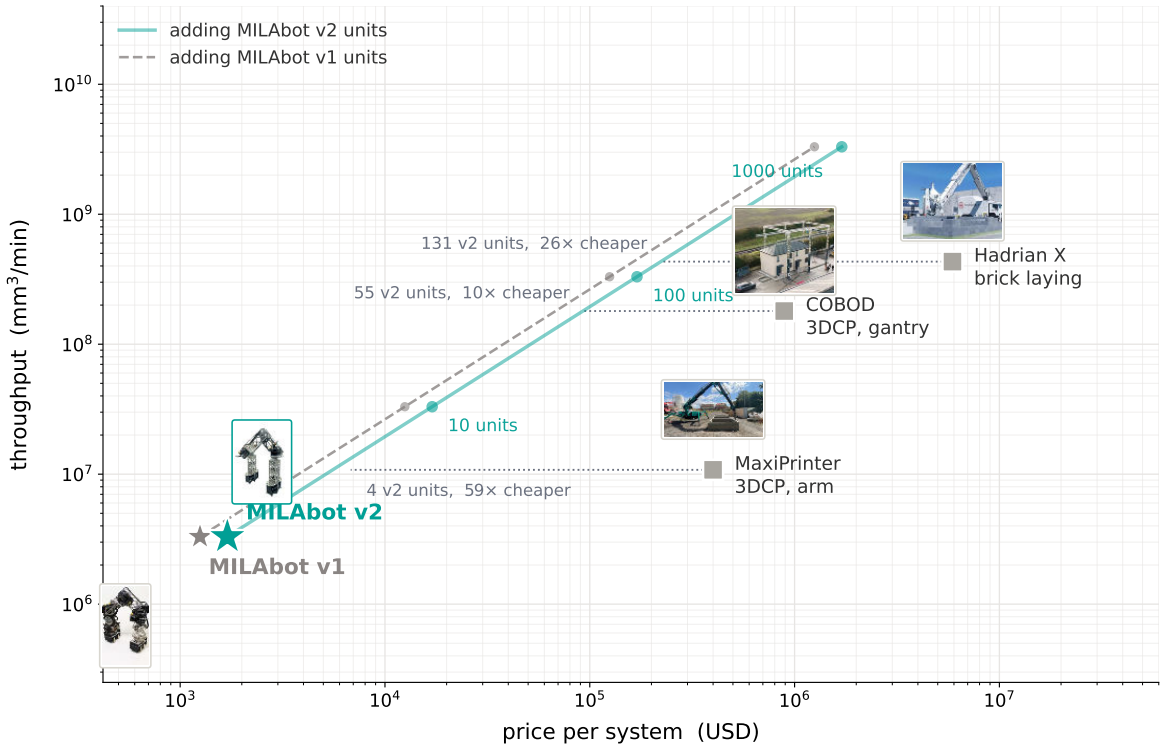


Figure 6.10: Throughput against price for autonomous construction systems, after [16], with MILAbot v2 added. Throughput uses that work's formulas, and the MILAbot v2 point is timed on the physical machine: at the same moves per cell as v1 and a marginally quicker move, the 1.54× larger voxel puts it at 1.62× v1's volume rate (Section 6.5.1), which is also why the v2 line lies to the left of v1's. The teal line shows the effect of buying more MILAbots, where N units cost $N \times 1700$ USD and deliver N times the throughput, and dotted leaders give that count and the resulting cost ratio. Prices for the three commercial systems are order-of-magnitude figures recovered from the cost ratios reported in [16], not quoted list prices.

Everything that requires the physical robot is missing from the results above, and Chapter 7 takes up what those gaps mean for the claims this chapter makes.

Chapter 7

Limitations and Future Work

7.1 No Full-Scale Build on Site

No structure has been assembled at full scale, and no fleet has built anything outside simulation. Two questions follow that the thesis cannot answer. The first is how reliable the link to the robot is over a long run. Every mirroring session reported here lasts minutes, whereas a build at architectural scale runs for hours, during which the bridge, the WiFi connection and the onboard controller all have to stay up and in step with the plan. The second is how reliable the machine itself is over that same horizon, covering joint drift, gripper wear, snap-fit engagement after many insertion cycles, and power. A plan certified stable and kinematically feasible still assumes a robot that keeps working.

→ Future work:

a full-scale build on site, instrumented for uptime and repeatability over a long run rather than for a single successful placement, with the hardware experiments of Section 5.3.3 as the first step toward it.

7.2 Robot Weight Outside the Movement Decision

The stability model carries the weight of the robots and injects it through the feet (Section 3.1.5), but neither the sequence planner, which reasons over placements, nor the stance planner, which reasons over reachable stances, asks whether a robot standing on a particular cell brings the partially built structure down. The obstacle is cost, since a stability solve is fast enough to certify a finished sequence and far too slow to run at every step of every walk, so the twin currently relies on the structure having been certified under a robot load and treats it as safe to walk on anywhere. Closing

that gap would make the certificate hold continuously rather than only at the placements. The components for it already exist, since the solver accepts a robot load at an arbitrary stance and the twin knows where every foot stands at every step. What is missing is the computation to call one from the other at that rate, which is the one part of this system that wants a larger machine than the laptop everything else was built to run on.

A second gap sits underneath it. The model is quasi-static, so it certifies that a set of contact forces balancing the structure exists and says nothing about the impact of seating a block, the vibration of a robot walking, or the compliance of the lattice under a moving load. A sequence certified stable can therefore still fail dynamically if a placement is executed roughly, which makes the quasi-static assumption the single strongest caveat on every verdict in this thesis.

→ Future work:
running the platform on hardware sized for it, so that a solve carrying the robot load can be issued at every step of every walk rather than only at the placements, an incremental check that makes those solves cheaper still, for instance a warm-started solve restricted to the cells around the robot, and a dynamic extension bounding the impulse a placement or a step is allowed to deliver.

7.3 Site and Fleet Assumptions

The planner works against an idealised site. Lattice cells coincide wherever two fronts meet, and the fleet building on them has its own material, a division of labour fixed in advance, and no neighbours. None of the four holds on the ground.

Almost every structure in this thesis closes a seam. A bridge deck meets in the middle, a wall returns to its corner, and a roof course lands on two walls raised separately, so two parts of the build that grew from different places have to meet on the same lattice cells. The stability model and the sequence planner both work on the ideal lattice, where those cells coincide by definition.

On hardware they coincide only if the build started from an accurately located and anchored base. The alignment features absorb roughly half a pitch of placement error per block (Section 1.2), but the correction is local and taken relative to the block below. It removes neither a global offset in the ground course nor the error accumulated over a long run of blocks placed away from a datum, so two fronts that each drift by less than the per-block tolerance can still be a full cell apart where they meet, and there the closing block does not fit.

That assumption carries more weight in this thesis than the results make visible. The cantilevers of the hardware experiments grow from one anchored root and never close against anything, whereas the simulated structures for which the stability model matters most, the bridges, the arche, the enclosures and the closing courses of the tiny house, all depend on a seam meeting. Their build times and their stability verdicts therefore assume a registration accuracy that has not been demonstrated

on hardware.

The remaining three assumptions are the fleet coordinator's of Section 5.1.2, and concern how the work divides. The first is that every robot has its own feed. Blocks are assigned to the nearest stock stripe, so two robots never contend for the same block and no stripe runs short while another sits full. A site is more likely to have fewer feed points than robots, one elevator serving a face, or stock replenished on a schedule the planner knows nothing about. Under those conditions the partition of the work is no longer a partition of the material.

The second is that the partition is complete and static. Work is divided once, geometrically, and each robot then places its queue strictly in order. Nothing rebalances when one robot's region turns out harder than another's, when a placement fails, or when a robot stops. The gated waves keep partitions from interfering by deferring work rather than by moving it, so a stalled region stalls the whole course.

The third is that structures are independent. At village scale each structure carries its own fleet and its own climb column, and no robot crosses to a neighbouring build or shares material with it. That is a modelling convenience, and it is the reason village-scale wall-clock in Section 6.4 is a sum of independent builds rather than a coordination result.

→ Future work:

an anchoring and datum strategy for the ground course with a hardware measurement of the error accumulated over a long run of placements, a seam-aware planner that either grows a structure from a single front or reserves an adjustable closure block where two fronts meet, a coordinator that models feed points as shared resources with finite capacity, dynamic reassignment of work when a region stalls or a robot drops out, and fleets that span structures rather than being bound to one.

7.4 Movement Catalog Bounds the Geometry

The locomotion catalog and the placement gesture families cover the benchmark suite and built every structure in this thesis. They are a fixed vocabulary rather than a complete one. A more geometrically involved target, with tighter reentrant corners, deeper overhangs, or seats the current gestures cannot approach from above, would call for placement families that do not yet exist, and the planner would report those cells unreachable rather than find another way to serve them. Which geometries the catalog can and cannot reach has not been characterised.

→ Future work:

a feasibility study mapping target geometry to the gestures it requires, new locomotion and placement families for the cases that fail, and an optimisation of the catalog itself, since the number of moves per placed block is the quantity that sets build time.

7.5 Deployment Where the Supply Chain Is the Constraint

The site this pipeline is aimed at is remote (Section 1.3), and a month spent working on the ground in Bhutan changed what that means for the planning layer. The system as presented assumes that voxels arrive in the quantity and the orientation a plan asks for, that a damaged part is replaced, and that a robot that breaks is repaired. Each of those assumes a supply chain, and where the nearest replacement for a bought component is weeks away, none of them is free.

Three consequences follow for the planner, and none is addressed in this thesis. The first is that material becomes a hard budget rather than a cost term. Support voxels, whether propped or grown, are counted and reported in Section 6.1 but never constrained, and a planner that may not exceed the stock on hand is a different planner. The second is that the block catalog is itself a supply decision. Sixteen block types are cheap to cut in a fab lab and expensive to hold as an inventory, so a planner able to trade plan quality against the number of distinct part types is worth more on site than one that assumes the whole catalog. The third is repairability. The argument for discrete assembly is that a damaged structure is repaired by exchanging cells rather than replaced, and nothing in this pipeline plans a disassembly, even though the stability model that certifies a build is the same model a disassembly would have to satisfy in reverse.

A fourth consequence sits upstream of the planner and was the binding one in practice. Over the month on site only part of the ordered stock arrived, and the robot build could not be finished there for want of components that were still in transit. Lead time, not fabrication capacity, set the pace of the visit, since the parts that had to be bought in governed the schedule while the parts that could be made locally did not. Deployment therefore has to be planned well in advance of arrival, with long-lead components ordered against the build schedule rather than against the visit, and with as much of the robot as possible brought into the set of parts a local fab lab can produce. A machine that can be rebuilt on site from locally made parts is worth more than one that is faster to assemble from parts that take weeks to reach it.

→ Future work:
planning under a material budget, a catalog term that prefers fewer distinct part types, disassembly and repair sequences certified by the same stability model, a robot design whose long-lead bought components are minimised in favour of locally fabricable ones, and a deployment study on site with locally fabricated voxels and locally sourced stock.

Figure 7.1 is what that constraint looks like in practice. Everything in the photographs was made on site, with the voxel face parts printed at the fab lab, the webs cut from local plywood, and the MILAbot's own structural plates and gripper components machined and printed alongside them. A system that can be fed this way is a different proposition from one that ships parts in, and it is the case the planner should be designed against.

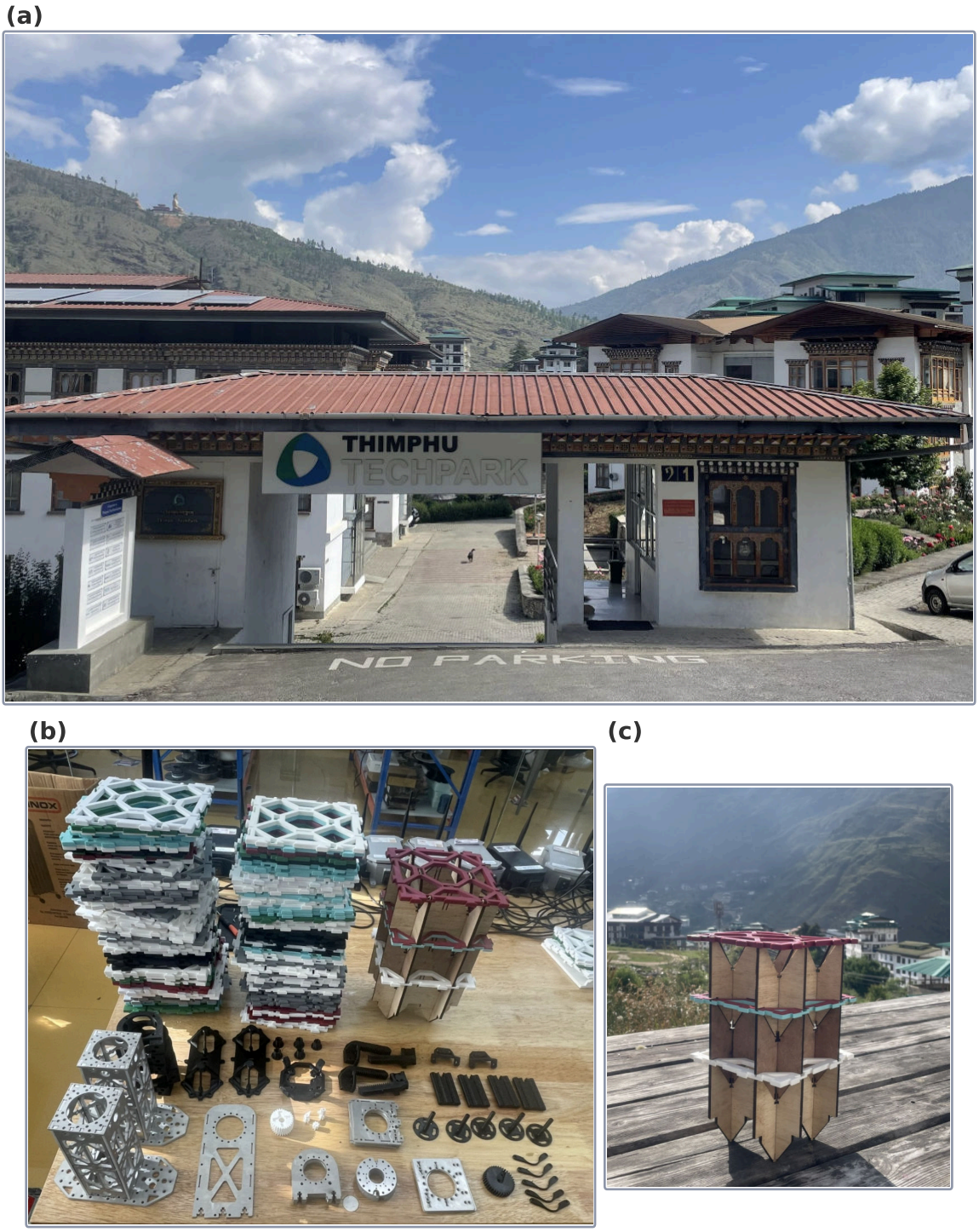


Figure 7.1: On-site fabrication in Bhutan. **(a)** the Thimphu TechPark, where the work was carried out. **(b)** voxel face parts and robotics components built during the stay, laid out on the bench. **(c)** a voxel assembled on site, standing against the valley. *Photographs: this thesis.*

Chapter 8

Conclusion

This thesis asked what voxel-based robotic construction requires in order to be *reliable*, meaning that what is planned is guaranteed to stand, step by step, under the weight of its own builders. It asked in the same terms what such construction requires in order to be *deployable*, meaning that a plan carries unchanged from an input mesh through simulation to the physical machine, at the scale of many structures built in parallel. The answer is the claim the introduction opened with. Construction becomes reliable when planning is grounded in a digital twin faithful to both the physics of the structure and the capabilities of the robot, and the six contributions are what that grounding turned out to require.

The stability model returns a verdict per block rather than one global number a planner cannot act on, and it carries the weight of the robots themselves (C1). On top of it, a definition of buildability separates a structure that stands when finished from one that stands at every step, and the sequence planner treats the physics as a constraint on what it may do rather than as a check applied afterwards, so the orders it returns are sound by construction (C2). Where no such order exists, a repair loop grows the geometry additively until the unmodified planner covers the original shape (C3). The twin reproduces how the current machine walks, reaches and places, with every motion a plan may call on validated against the real kinematics first (C4). The pipeline carries all of it from an input mesh to a fleet building a site, under the constraints the physical machine works under (C5), and a live bridge drives the robot from that same plan (C6).

In practice all of it became one tool. The analyzer and the simulator are a single browser-based platform, so a mesh is voxelised, coloured block by block by the stability analysis, patterned or propped, evolved, planned and then built by a fleet without touching the code underneath. Training a planner for a target is one button on a laptop rather than a job on a cluster. That the numbers of Chapter 6 came from the same code path a user runs, rather than from a separate experimental harness, is what makes them worth reporting.

The claim holds as far as it has been tested. On hardware the bridge drove the physical MILAbot v2 through cantilevered builds, the machine placing overhanging geometry under a plan the twin produced and the analyzer certified, on the one geometry where a purely geometric sequencer stalls mid-build. In simulation the same pipeline carried targets two orders of magnitude larger, up to a house of nearly ten thousand voxels, and sites on which several structures are raised at once by their own fleets.

What has not been shown is that agreement holding at scale. The hardware structures are a few courses built once under supervision, while the structures for which the stability model matters most have only ever been built in the twin. Running the pipeline on the machine at the scale the simulation already reaches is the next step, and Chapter 7 sets out what it would take.

The argument extends beyond this particular system. Discrete robotic construction has advanced so far by making robots more capable and materials more clever, and this thesis argues that the binding constraint is neither. It is *certification*, knowing before a robot climbs onto a half-built bridge that the bridge and the plan will hold it. Once the certificate exists, everything downstream inherits it, from planning and repair to the buildings of programmes like Gelephu's, where fleets of small robots and local materials are infrastructure rather than demonstration. Figure 8.1 closes on that thought, showing three structures this pipeline plans and certifies as the simulator produces them and as they would read built in a Bhutanese vernacular.

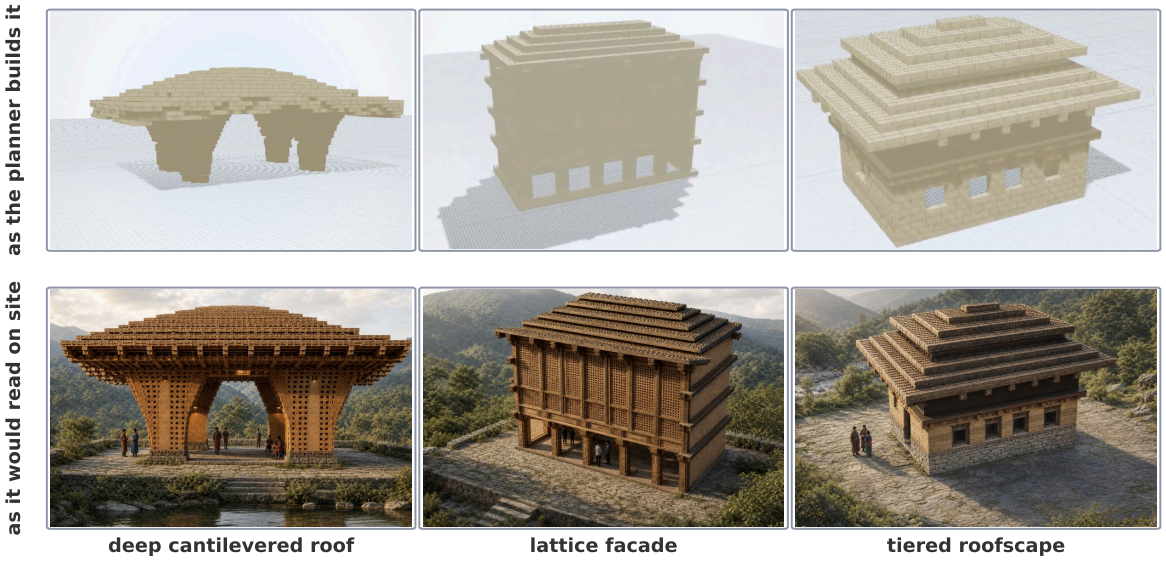


Figure 8.1: Three structures planned and certified by this pipeline, shown twice. Top: the voxel model the planner works on and the fleet builds. Bottom: the same geometry rendered in the Bhutanese vernacular a programme like Gelephu calls for, with deep cantilevered eaves, lattice facades, and tiered roofscapes. The architectural language and the machine input are one object, since every motif that reads as ornament in the bottom row is an overhang in the top row. Both rows are this project's own output, and neither is an image of Gelephu Mindfulness City.

Bibliography

[1] Bjarke Ingels Group. *Gelephu Mindfulness City*. Masterplan description and figures; the project's own material is cited alongside as [23]. 2023. URL: https://world-architects.com/en/big-bjarke-ingels-group-valby-copenhagen/project/gelephu-mindfulness-city (visited on 07/30/2026).

[2] Holcim Foundation. *Gelephu Mindfulness City: Holcim Foundation Awards 2025*. 2025. URL: https://www.holcimfoundation.org/projects/gelephu-mindfulness-city (visited on 07/30/2026).

[3] Richard A. Buswell, W. R. Leal de Silva, Scott Z. Jones, and Justin Dirrenberger. "3D Printing Using Concrete Extrusion: A Roadmap for Research". In: *Cement and Concrete Research* 112 (2018), pp. 37–49.

[4] Steven J. Keating, Julian C. Leland, Levi Cai, and Neri Oxman. "Toward Site-Specific and Self-Sufficient Robotic Fabrication on Architectural Scales". In: *Science Robotics* 2.5 (2017), eaam8986. DOI: 10.1126/scirobotics.aam8986.

[5] FBR Ltd. *Hadrian X: Automated Construction Robot*. URL: https://www.fbr.com.au/view/hadrian-x (visited on 08/01/2026).

[6] Kathrin Dörfler, Timothy Sandy, Markus Giftthaler, Fabio Gramazio, Matthias Kohler, and Jonas Buchli. "Mobile Robotic Brickwork: Automation of a Discrete Robotic Fabrication Process Using an Autonomous Mobile Robot". In: *Robotic Fabrication in Architecture, Art and Design 2016*. Springer, 2016, pp. 204–217.

[7] Constructions-3D. *MaxiPrinter: Mobile Concrete 3D Printer*. URL: https://www.constructions-3d.com/en/maxiprinter (visited on 07/30/2026).

[8] Roadstone. *Roadstone Develops Concrete for 3D Construction Printing*. Feb. 2024. URL: https://www.roadstone.ie/news/roadstone-develops-concrete-3d-construction-printing (visited on 07/30/2026).

[9] Kirstin H. Petersen, Nils Napp, Robert Stuart-Smith, Daniela Rus, and Mirko Kovac. "A Review of Collective Robotic Construction". In: *Science Robotics* 4.28 (2019), eaau8479. DOI: 10.1126/scirobotics.aau8479.

[10] Justin Werfel, Kirstin Petersen, and Radhika Nagpal. "Designing Collective Behavior in a Termite-Inspired Robot Construction Team". In: *Science* 343.6172 (2014), pp. 754–758.

[11] Ketao Zhang, Pisak Chermprayong, Feng Xiao, et al. “Aerial Additive Manufacturing with Multiple Autonomous Robots”. In: *Nature* 609.7928 (2022), pp. 709–717. DOI: 10.1038/s41586-022-04988-4.

[12] Christine E. Gregg, Damiana Catanoso, Olivia I. B. Formoso, et al. “Ultralight, Strong, and Self-Reprogrammable Mechanical Metamaterials”. In: *Science Robotics* 9.86 (2024), eadi2746. DOI: 10.1126/scirobotics.adi2746.

[13] Amira Abdel-Rahman, Christopher Cameron, Benjamin Jenett, Miana Smith, and Neil Gershenfeld. “Self-Replicating Hierarchical Modular Robotic Swarms”. In: *Communications Engineering* 1.35 (2022).

[14] Kenneth C. Cheung and Neil Gershenfeld. “Reversibly Assembled Cellular Composite Materials”. In: *Science* 341.6151 (2013), pp. 1219–1221.

[15] Benjamin Eric Jenett. “Discrete Mechanical Metamaterials”. PhD thesis. Massachusetts Institute of Technology, 2020.

[16] Paul Richard. “Workflows for a Scalable Lattice-Based Robotic Assembly”. Carried out at the MIT Center for Bits and Atoms. MA thesis. EPFL, 2025.

[17] Benjamin Jenett, Sam Calisch, Daniel Cellucci, Nick Cramer, Neil Gershenfeld, Sean Swei, and Kenneth C. Cheung. “Digital Morphing Wing: Active Wing Shaping Concept Using Composite Lattice-Based Cellular Structures”. In: *Soft Robotics* 4.1 (2017), pp. 33–48.

[18] Nicholas B. Cramer, Daniel W. Cellucci, Olivia B. Formoso, Christine E. Gregg, Benjamin E. Jenett, Joseph H. Kim, Martynas Lendraitis, Sean S. Swei, Greenfield T. Trinh, Khanh V. Trinh, and Kenneth C. Cheung. “Elastic Shape Morphing of Ultralight Structures by Programmable Assembly”. In: *Smart Materials and Structures* 28.5 (2019), p. 055006. DOI: 10.1088/1361-665X/ab0ea2.

[19] Benjamin Jenett and Kenneth C. Cheung. “BILL-E: Robotic Platform for Locomotion and Manipulation of Lightweight Space Structures”. In: *25th AIAA/AHS Adaptive Structures Conference*. 2017.

[20] Benjamin Jenett, Amira Abdel-Rahman, Kenneth Cheung, and Neil Gershenfeld. “Material–Robot System for Assembly of Discrete Cellular Structures”. In: *IEEE Robotics and Automation Letters* 4.4 (2019), pp. 4019–4026. DOI: 10.1109/LRA.2019.2930486.

[21] Miana Smith, Amira Abdel-Rahman, and Neil Gershenfeld. “Self-Reconfigurable Robots for Collaborative Discrete Lattice Assembly”. In: *IEEE International Conference on Robotics and Automation (ICRA)*. Yokohama, Japan, 2024. URL: https://cba.mit.edu/docs/papers/24.05.ICRA.pdf.

[22] Miana Smith, Paul Arthur Richard, Alexander Htet Kyaw, and Neil Gershenfeld. “Hierarchical Discrete Lattice Assembly: An Approach for the Digital Fabrication of Scalable Macroscale Structures”. In: *ACM Symposium on Computational Fabrication (SCF)*. 2025. URL: https://cba.mit.edu/docs/papers/25.11.Smith.SCF.pdf.

[23] Gelephu Mindfulness City. *Gelephu Mindfulness City: Project Gallery*. Official project imagery; masterplan by BIG with Arup and Cistri. Gelephu Mindfulness City, Royal Government of Bhutan. 2026. URL: https://gmc.bt/gallery (visited on 07/30/2026).

[24] Kuensel. *Bhutan Partners with MIT to Pioneer Robotic Construction for GMC*. Agreement between Druk Holding and Investments and the MIT Center for Bits and Atoms for voxel-based robotic construction in support of the Gelephu Mindfulness City, led on the Bhutanese side by the Jigme Namgyel Wangchuck Super Fab Lab. Aug. 2025. URL: https://kuenselonline.com/news/bhutan-partners-with-mit-to-pioneer-robotic-construction-for-gmc (visited on 07/30/2026).

[25] Miana Smith. *MILAbot V2 Robotic Assembly*. MIT Center for Bits and Atoms. Design and control repository for the successor generation described in [22]. 2025. URL: https://gitlab.cba.mit.edu/miana/milabot-v2-assembly.

[26] Ruixuan Liu, Kangle Deng, Ziwei Wang, and Changliu Liu. "StableLego: Stability Analysis of Block Stacking Assembly". In: *IEEE Robotics and Automation Letters* 9.11 (2024), pp. 9383–9390.

[27] Miana Smith, Jack Forman, Amira Abdel-Rahman, Sophia Wang, and Neil Gershenfeld. "Voxel Invention Kit: Reconfigurable Building Blocks for Prototyping Interactive Electronic Structures". In: *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*. Reduced-order beam-model FE for voxel lattices; the inspiration for this thesis's FE cross-check. ACM, 2025. DOI: 10.1145/3706598.3713948.

[28] Emily Whiting, John Ochsendorf, and Frédo Durand. "Procedural Modeling of Structurally-Sound Masonry Buildings". In: *ACM Transactions on Graphics* 28.5 (2009), 112:1–112:9. DOI: 10.1145/1618452.1618458.

[29] Romain Prévost, Emily Whiting, Sylvain Lefebvre, and Olga Sorkine-Hornung. "Make It Stand: Balancing Shapes for 3D Fabrication". In: *ACM Transactions on Graphics* 32.4 (2013), 81:1–81:10. DOI: 10.1145/2461912.2461957.

[30] Mario Deuss, Daniele Panozzo, Emily Whiting, Yang Liu, Philippe Block, Olga Sorkine-Hornung, and Mark Pauly. "Assembling Self-Supporting Structures". In: *ACM Transactions on Graphics* 33.6 (2014), 214:1–214:10. DOI: 10.1145/2661229.2661266.

[31] Yunsheng Tian, Jie Xu, Yichen Li, Jieliang Luo, Shinjiro Sueda, Hui Li, Karl D. D. Willis, and Wojciech Matusik. "Assemble Them All: Physics-Based Planning for Generalizable Assembly by Disassembly". In: *ACM Transactions on Graphics* 41.6 (2022), 278:1–278:11. DOI: 10.1145/3550454.3555525.

[32] Yunsheng Tian, Karl D. D. Willis, Bassel Al Omari, Jieliang Luo, Pingchuan Ma, Yichen Li, Farhad Javid, Edward Gu, Joshua Jacob, Shinjiro Sueda, Hui Li, Sachin Chitta, and Wojciech Matusik. "ASAP: Automated Sequence Planning for Complex Robotic Assembly with Physical Feasibility". In: *IEEE International Conference on Robotics and Automation (ICRA)*. 2024. arXiv: 2309.16909.

[33] Niklas Funk, Georgia Chalvatzaki, Boris Belousov, and Jan Peters. “Learn2Assemble with Structured Representations and Search for Robotic Architectural Construction”. In: *Proceedings of the 5th Conference on Robot Learning (CoRL 2021)*. Vol. 164. Proceedings of Machine Learning Research. Conference held November 2021; proceedings published 2022. 2022, pp. 1401–1411.

[34] Ruixuan Liu, Alan Chen, Weiye Zhao, and Changliu Liu. “Physics-Aware Combinatorial Assembly Sequence Planning using Data-free Action Masking”. In: *IEEE Robotics and Automation Letters* 10.5 (2025), pp. 4882–4889. DOI: 10.1109/LRA.2025.3555074. arXiv: 2408.10162 [cs.RO].

[35] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. *Proximal Policy Optimization Algorithms*. arXiv:1707.06347. 2017.

[36] Shengyi Huang and Santiago Ontañón. “A Closer Look at Invalid Action Masking in Policy Gradient Algorithms”. In: *International FLAIRS Conference*. 2022.

[37] Pablo Funes and Jordan Pollack. “Evolutionary Body Building: Adaptive Physical Designs for Robots”. In: *Artificial Life* 4.4 (1998), pp. 337–357. DOI: 10.1162/106454698568639.

[38] Ava Pun, Kangle Deng, Ruixuan Liu, Deva Ramanan, Changliu Liu, and Jun-Yan Zhu. “Generating Physically Stable and Buildable Brick Structures from Text”. In: *IEEE/CVF International Conference on Computer Vision (ICCV)*. Method named BrickGPT. 2025. arXiv: 2505.05469.

[39] Hanne Kekkonen. “Exploring Mathematics with Curvagon Tiles”. In: *Proceedings of Bridges 2022: Mathematics, Art, Music, Architecture, Culture*. arXiv:2208.00419. 2022, pp. 183–190. ISBN: 978-1-938664-42-7. URL: https://archive.bridgesmathart.org/2022/bridges2022-183.html.

This project used Claude Code and Cursor for assistance in developing and refining the code, ensuring efficient implementation and optimization of the algorithms. These tools served as a support during the development process, as well as for grammar checks and structure enhancement of this report.

Appendix A

Stability Model Details

This appendix collects the constants the stability model of Section 3.1 runs on. All of them live in a single source of truth, `shared/config/solver_config.py`, which both the analyzer backend and the planner import.

A.1 Model Constants

Quantity	Value	Notes
Vertical snap fit, fresh	40 N	pull-apart between stacked cells, per cell; a 2 × 2 face carries four. The design default
Vertical snap fit, worn	8 N	after a few insert and remove cycles the wood settles here. Assembled structures stay seated, so this is the conservative bound, and the one the regime argument of Section 3.1.1 uses
Side dovetail	~250 N	2 × 2 to 2 × 2. Reference only: side ties are not treated as design-limiting and the solver leaves them effectively unbounded
Block crushing load $C_{\max}$	10–14 kN	pure compression through a block; the lower bound is the design value
Bearing strength	1.78 MPa	$C_{\max}$ spread over one 75 × 75 mm face, 2.49 MPa at the upper crushing bound. Used only as the allowable stress in the FE cross-check, which sees a continuum and has no joints to check

Table A.1: Measured capacities of the MILAbot v2 block. Fresh and worn snap-fit values differ by a factor of five, and which one applies depends on how often the joint has been cycled, so the model exposes $T_{\text{snap}}^{\max}$ as a parameter rather than fixing it (Figure 3.6).

Quantity	Value	Notes
Lattice pitch a	75 mm	the only supported pitch
Density ρ	137 kg m^{-3}	effective and fixed; per-cell mass scales with a^3
Mass per cell	0.0578 kg	ρa^3, a weight of 0.566 N
Gravity g	9.8 m s^{-2}	
Robot mass	4.5 kg	44.1 N, injected through the feet (Section 3.1.5)
Ground contact plane	$z_c - 2h$	h the half-cell, cell-relative

Table A.2: Geometry and material constants. One robot weighs 78 cell weights, which is why its load dominates the verdict in every assembly state it appears in.

A.2 Solver Settings

The existence problem of Section 3.1.6 is solved in Gurobi. The objective weights the equilibrium residual against two regularizers: $\alpha = 10^{-3}$ penalizes excessive vertical loading, which is what produces the per-block stress visualization, and $\beta = 10^{-6}$ regularizes toward minimal-force solutions. Both are small enough that a nonzero residual always dominates the objective.

Side and lateral snap variables are held at a finite 10^6 N rather than left unbounded, a value far above any realistic load but low enough to keep the program bounded. The strict-inequality variant of the capacity constraint enforces $f < T$ as $f \le T - \varepsilon$ with $\varepsilon = 10^{-4}$ N.

Appendix B

Training Details

The planner is `MaskablePPO` from `sb3_contrib` with a `MaskableMultiInputActorCriticPolicy`, and the action mask is supplied per step by the environment.

B.1 Hyperparameters

Table B.1 lists the defaults in `asp-training/core/asp_training_hyperparams.py`. These are the values the interface's *Train* button uses, and the values behind every training cost reported in Section 6.2.1.

B.2 Observation and Feature Extractor

The policy sees a 16^3 workspace as a stack of seven channels (`asp-training/core/observation_feature_extractor.py`), namely current occupancy, target occupancy, normalised per-column top height broadcast over z, per-layer completion fraction also broadcast, the previous layer's projection, the per-layer remaining fraction, and an active-layer hint. The first four channels carry the geometry, while the last three are the pattern channels added by the observability improvements of Section 3.3.3.

The extractor is small on purpose, because the branching factor is small:

```
Conv3d(7, 16, k=3, s=2, p=1) → GELU
Conv3d(16, 32, k=3, s=2, p=1) → GELU
AdaptiveAvgPool3d(1) → Flatten
‖ Linear(7, 64) → MultiheadAttention(64, 4 heads) → LayerNorm(64) →
```

Parameter	Value	Note
PPO		
`n_envs`	4	parallel environments
`n_steps`	128	rollout length per environment
`batch_size`	64	
`n_epochs`	3	
`gamma`	0.95	
`gae_lambda`	0.95	
`clip_range`	0.2	
`vf_coef`	1.0	
`learning_rate`	2×10^{-4}	linearly annealed
`learning_rate_end`	5×10^{-5}	
`ent_coef`	0.35	annealed; high early exploration
`ent_coef_end`	0.08	
Budget and episode		
`num_iters`	20 000	timesteps, scaled by target size
`max_outer_iters`	60	outer retrain loops; no cap is imposed by the interface
`outer_patience`	2	outer loops without improvement
`DEFAULT_MAX_STEPS`	180	episode horizon, large targets
`MAX_STEPS_SMALL_TARGET`	60	16^3 bench, small cantilever
Network		
`features_dim`	96	feature-extractor output width
`mlp_arch`	(128, 64)	actor/critic head, `Tanh`
`use_pattern_channels`	true	7 input channels, not 4
`use_layer_attention`	true	adds the attention branch

Table B.1: PPO and network defaults. Both the episode horizon and the timestep budget are scaled per target by $\sqrt{n_{\text{voxels}}/256}$ off a 256-voxel reference and floored at the base value.

```
Linear(64, 64)
concat(32, 64) → Linear(96, 96) → GELU
```

The two convolutions take the 16^3 grid to 8^3 and then 4^3 before global pooling, while the parallel branch attends over layers.

B.3 Reward

The reward is dominated by coverage. Table B.2 lists the weights, taken from `asp-training/core/assembly_env.py`.

Term	Weight	Fires when
Fill fraction	1.0	every step, on coverage of the original target
Outside-target penalty	1.5	per scaffold or outside cell, as a fraction of the target
Layer completion	+2.0	first time a z-layer is fully covered
Stagger, single	+0.018	a staggered block is placed
Stagger, chained	+0.05	a chained stagger is placed
Stagger, triple	+0.09	a triple stagger is placed
Potential shaping	0.05	on $\log(1 + \lvert\text{legal actions}\rvert)$, in the sense of Ng et al.
Terminal success	+5.0	full coverage reached
Terminal time bonus	+0.5	scaled by the unused fraction of the horizon
Truncation penalty	10.0	scaled by the coverage shortfall
Dead-end penalty	6.0	no legal action remains, weighted $0.25 + 0.75f$ in the fill fraction f

Table B.2: Reward terms and their weights. Stagger bonuses are scaled at runtime by target size, so they stay proportionate on large structures. The dead-end penalty deliberately grows with coverage, since a policy that strands itself at 90 % has wasted more than one that strands itself at 10 %.

B.4 Action Mask

A placement is legal only if it survives all six predicates below, all implemented in `valid_action_mask()` in `assembly_env.py` with matching native and Numba fast paths.

1. **No overlap.** No cell of the candidate is already filled.
2. **Column clear.** Nothing is already stacked above the block's own top, so the arm can come straight down.
3. **Layer order.** The z-layer immediately below the block's lowest cell is complete.
4. **Stagger safety.** For a block spanning two courses, no exposed top column that its own bottom footprint does not cover may sit over an unfilled target cell, which would seal that cell off.
5. **Support.** At least one cell rests on the floor or on a filled cell directly below.
6. **Residual guard.** The placement must not leave an unfillable one-cell-wide slit of target behind it.

Appendix C

Algorithms

Each of the four algorithms below is given in the form in which it is actually implemented, with its source file named.

C.1 Deterministic Target Repair

Section 3.4 grows a non-buildable target into a buildable one. Priority order matters, because grounding a floating base can remove an overhang violation for free, while the reverse is not true.

The violation types are given below in the priority order the loop resolves them, each with the mutation it triggers:

1. **Floating base:** a column whose lowest occupied voxel sits at $z \in [1,2]$ is filled straight down to the floor.
2. **Odd solid layer:** a solid rectangular slice with an odd side is padded along one full edge, trying $+x$, $-x$, $+y$, $-y$ in turn, up to 32 iterations, until both spans are even.
3. **Odd bounding box:** a non-solid slice is treated the same way, except that the box is extended by copying the adjacent edge's occupancy rather than filling a blind strip, so nothing protrudes.
4. **Single-layer overhang:** a roof-exposed cell with nothing supporting it to depth d causes that layer's own footprint to be extruded down d layers, again as a shape copy.
5. **Thin ribbon:** a cell with no same-layer neighbour on one axis is widened perpendicular to the missing axis.
6. **Isolated cell:** a 1×1 island is removed, together with every other such cell in one pass.

Algorithm 1 Repair a target until it is buildable (`stability_evolve/pipeline.py`)

procedure REPAIRUNTILOK(V, *maxSteps* = 256, $d = 1$)
  for $i \leftarrow 1$ **to** *maxSteps* **do**
    $\mathcal{V} \leftarrow$ COLLECTVIOLATIONS(V, d) ▷ ordered by priority
    if $\mathcal{V} = \emptyset$ **then**
      return V ▷ buildable
    end if
    $V', changed \leftarrow$ MUTATE(V, $\mathcal{V}_1$, d)
    if $\neg\, changed$ **or** $V' = V$ **then**
      return V ▷ stuck, report rather than loop
    end if
    $V \leftarrow V'$
  end for
  return V ▷ budget exhausted
end procedure

procedure REPAIRPROGRESSIVE(V, $D = (1,2,3)$)
  for $d \in D$ **do**
    $V \leftarrow$ REPAIRUNTILOK(V, 256, d) ▷ each pass only deepens what is still flagged
  end for
  return V
end procedure

Every mutation except the last is purely additive. The horizontal reach allowance is set to zero, so a cell with nothing directly beneath it is never assumed reachable from a neighbouring column.

C.2 Stability Solve

The verdict of Section 3.1 is a linear program, one per structure state. Its variables are the contact forces at every cell interface, and its objective is the total equilibrium residual.

The regularisers $\alpha = 10^{-3}$ and $\beta = 10^{-6}$ are small enough that a nonzero residual always dominates, which is what makes the verdict a feasibility question rather than a cost comparison.

C.3 Placement Family Selection

Section 4.5 chooses which of the placement families executes a given block. The families are ranked by preference rather than scored, with side approaches tried first because they disturb the least airspace, then from-behind approaches, and finally the wide swings.

Algorithm 2 Stability verdict for one assembly state (`pipeline/backend/stablelego/runner.py`)

procedure ANALYSE(B, R, T, $C_{\max}$)
  assign each brick $b \in B$ a mass $\rho\, s^3 h\, w$ ▷ $\rho = 137\,\mathrm{kg/m^3}$, $s = 75\,\mathrm{mm}$
  for each robot $r \in R$ **do**
    split r's weight between its two feet by projecting its centre of mass onto the segment joining them, clamped to $[0, 1]$
  end for
  build the occupancy grid and the cell-to-brick map
  for each brick b, each occupied cell c of b **do**
    add contact-force variables at c's contact points ▷ three points inside the interlock
    add snap-fit tension variables on exposed top faces, bounded by T
    couple horizontal forces across brick-to-brick interfaces
  end for
  for each brick b **do**
    $\sum F_z = \sum f_{\uparrow} - \sum f_{\downarrow} - W_b - W_{\mathrm{robot},b}$ ▷ and likewise F_x, F_y
    $\sum M_1, \sum M_2$ from the moment arms of those forces
    introduce $|F_\bullet|$ and $|M_\bullet|$ by linearisation
  end for
  minimise $\sum_b \left(|F_x| + |F_y| + |F_z| + |M_1| + |M_2|\right) + \alpha \sum_b \max f_{\downarrow} + \beta \sum f_{\downarrow}$
  solve ▷ barrier method
  for each brick b **do**
    $res \leftarrow |F_x| + |F_y| + |F_z| + |M_1| + |M_2|$
    $m \leftarrow T - \max(\text{tension on any cell of } b)$; $u \leftarrow \text{compression}/C_{\max}$
    if $res > \varepsilon$ **or** $m \le 0$ **or** $u \ge 1$ **then**
      score(b) $\leftarrow 1$ ▷ unstable, record why
    else
      score(b) $\leftarrow \mathrm{clamp}(\max(1 - m/T,\, u)\,, 0, 1)$
    end if
  end for
  return stable $\Leftrightarrow$ no brick scored 1
end procedure

Column climb tiers are exempt from the there-and-back gate, since the robot is meant to stay on them.

C.4 Patch Decomposition and Symmetry Reuse

Targets larger than one 16^3 workspace are split, and the split exploits symmetry twice.

Only the unique patches in U are trained, while the rest are reconstructed by replaying a representative's policy and mirroring the result back across the planes in P. Two further redundancies are detected inside a patch and used the same way, namely a mirror axis of the patch's own occupied

Algorithm 3 Choose a placement family for one block (`robot-assembly/src/lib/assemblyBrickPlanner.ts`)

procedure PLACEBLOCK(b, *source*)
  $F \leftarrow$ preference-ordered placement families
  $E \leftarrow \emptyset$ ▷ families already ruled out
  while $E \neq F$ **do**
    $p \leftarrow$ PLANPLACEMENT(b, *source*, $F \setminus E$)
    if p exists **then**
      return p
    end if
    add the failing family to E
  end while
  if b is a support with nothing to stand beside **then**
    return the in-place from-behind drop ▷ the only family that bootstraps
  end if
  return FAIL ▷ reported, the sequence continues
end procedure

procedure VALIDATE(candidate stance s, family f)
  plan an A* walk to s; fail if none within the node and time budget
  fail if b would land on the robot's own standing feet
  fail if no validated gesture clip exists for (f, z, foot parity)
  fail if the robot cannot walk back to the stock stripe afterwards ▷ the there-and-back gate
  return pass
end procedure

Algorithm 4 Decompose a large target into trainable patches (`asp-training/patch/asp_patch_build_simple.py`)

procedure BUILDPATCHES(V, *chunk* = 16)
  $S, P \leftarrow$ SPLITBYSYMMETRYPLANES(V) ▷ every mirror plane
  if more than one chunk and the symmetry is fundamental **then**
    keep the chunk with the most voxels after clipping to the half-model ▷ the fundamental domain
  end if
  $Q \leftarrow$ tile that chunk into 16^3 patches, bottom-aligned
  sort Q symmetry-seed first; renumber $0 \ldots N-1$
  $U, G \leftarrow$ DEDUPE(Q) ▷ byte-identical local targets collapse to one
  return Q, U, G, P
end procedure

bounding box, requiring a span of at least two cells on the low half, and a vertical repeat period, requiring at least two repeats, so that a column can be trained once and stacked.

Appendix D

Reference Tables

D.1 Block Catalog

Table D.1 is the catalog the simulator and the executor place from, defined in `robot/robot-build/src/lib/brickCatalog.ts`. Each entry is a set of cell offsets, so a staggered block is not a special case in the code but simply a block whose upper course is offset from its lower one. Table D.2 is the smaller subset the planner trains on.

D.2 Benchmark Structures

Voxel counts in Table D.3 are read from the analyzer's own voxelisation at 75 mm (`asp-training/data/*/*_voxels.json`), so they are the grids the planner actually trains on rather than nominal figures.

Block	Cells	Description
Single course		
`2x2`	4	the unit block
`2x3`	6	
`2x4`	8	
Two courses, aligned		
`2x2x2`	8	
`2x3x2`	12	
`2x4x2`	16	
Two courses, staggered		
`2x2x2-o1`	8	upper course offset one cell
`2x3x2-o1-x`	12	offset one cell in x
`2x3x2-o1-y`	12	offset one cell in y
`2x3x2-o2-y`	12	offset two cells in y
`2x4x2-o1-x`	16	
`2x4x2-o1-y`	16	
`2x4x2-o2-y`	16	
`2x2x2-o1-xy`	8	offset on both axes
`2x3x2-o1-xy`	12	
`2x4x2-o1-xy`	16	
`3x2x2-o2-x`	12	defined in the catalog but not exposed in the interface

Table D.1: The placement catalog.

Id	Block	Role
0	`2x4`	flat
1	`2x3`	flat
2	`2x2`	flat
3	`2x4x2-o2-y`	single stagger
4	`2x8x2-o2-y-chain`	chained stagger
5	`2x12x2-o2-y-chain`	triple stagger
6	`3x2x2-o2-x`	cross-axis stagger

Table D.2: The seven blocks the policy chooses among.

Structure	Voxels	Role in this thesis
Cantilever, half	96	the headline dead-end and repair case
Cantilever, deck	100	second repair case
Simple bench	192	smallest full benchmark
Voxel cart	808	the four-robot fleet case
Single layer	864	flat-tiling reference
C-pod	1324	
Covered shelter	1568	holds the one stubborn patch
Archway	1728	near-identical size, an order of magnitude cheaper
Tiny house v2	2126	
Tiny house v1	2372	twelve easy patches
Enclosed bench	2992	largest single-window benchmark
Small bridge	6240	
Arche	6540	patch-wise evolution case
Bhutan house	9808	largest target attempted

Table D.3: Benchmark structures and their voxel counts at 75 mm.

Appendix E

Software and Reproduction

E.1 Repository

Every result in this thesis was produced by the code in a single repository, held at the MIT Center for Bits and Atoms:

`https://gitlab.cba.mit.edu/miana/milabot-v2-assembly`

It holds roughly 27 300 lines across some 82 source files in JavaScript, Python, HTML and CSS. The parts this thesis relies on are:

`pipeline/` The analyzer, covering voxelisation, pattern generation, and the stability backend of Section 3.1. The solver entry point is `pipeline/backend/stablelego/runner.py`, and the formulation is documented in `pipeline/stability_analysis_formulation.md`.

`asp-training/` The sequence planner of Section 3.3, containing the environment, feature extractor, hyperparameters, patch decomposition, and the target-evolution work of Section 3.4.

`robot/` The digital twin of Chapter 4 and the executor, covering kinematics, the locomotion catalog, path planning, placement families, the build sequencer, and the hardware bridge of Section 5.3.2.

`shared/` The single source of truth both sides import, holding the voxel pitch, grid conventions, the block catalog, and the solver constants of Appendix A.

E.2 Running the Pipeline

The analyzer's stability solve uses Gurobi and therefore needs a licence. The script `setup-env.sh` installs one and creates the virtual environment, while `start.sh` brings the whole system up on `localhost:5050`. From there the pipeline is the one described in Section 5.1.1.

Reproducing the training costs of Section 6.2.1 takes four commands, run from the repository root and then from `Thesis_Writing/`:

```
node asp-training/tools/voxelize_stls.mjs --out asp-training/data_bench
cd asp-training && python bench_asp_training_cost.py --all --budget-s 0
python measure_action_space.py
python3 scripts/make_asp_cost_fig.py
```

The first command voxelises the benchmark STLs with the analyzer's own voxeliser, so that the grids match what the interface trains on. The second trains the suite and writes `out/thesis_bench/bench_results.json`, the third measures the action-space funnel, and the fourth regenerates the figure. Training writes only under `asp-training/out/thesis_bench/`, so the checkpoints the pipeline uses are never touched.

Trained checkpoints are registered in `asp-training/MODELS.md`, where each benchmark structure maps to a recipe, an output directory and a checkpoint file, and patch-trained targets warm-start from the cantilever checkpoint.