Motion Planning and Task Planning

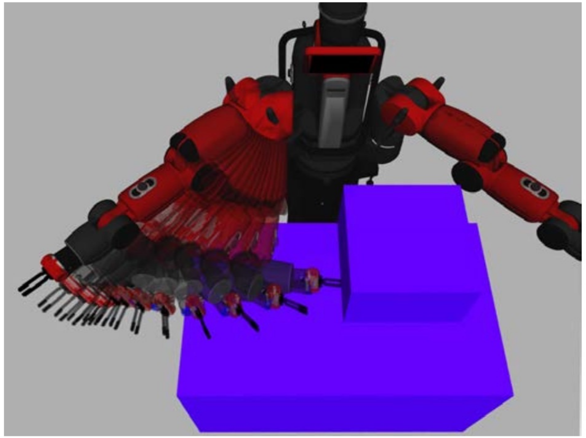
From Random Trees to Foundation Models

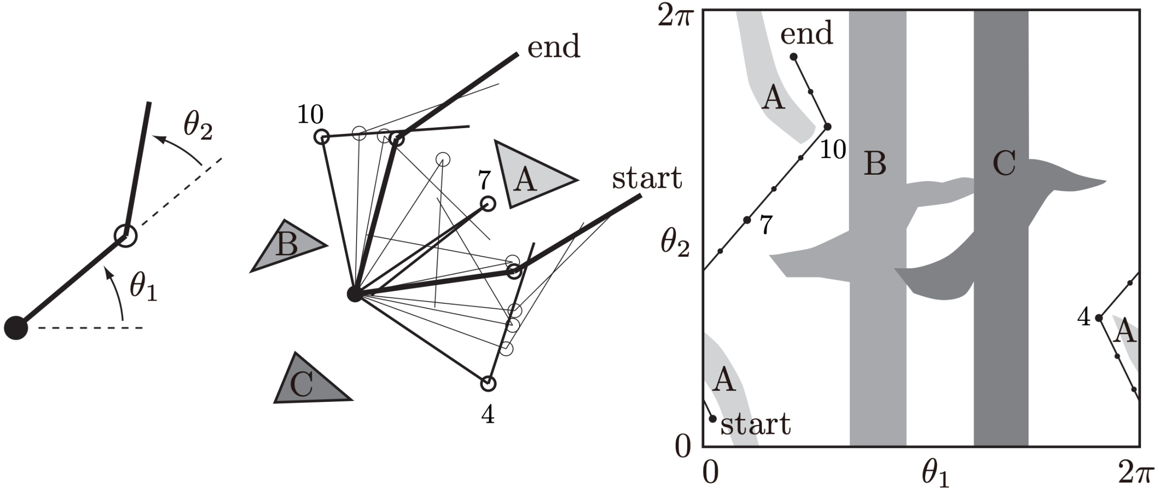
The Planning Question

Kinematics (Lecture 2) told us how to represent configurations.

Control (Lecture 3) told us how to track trajectories.

Now: how does the robot decide what trajectory to execute?

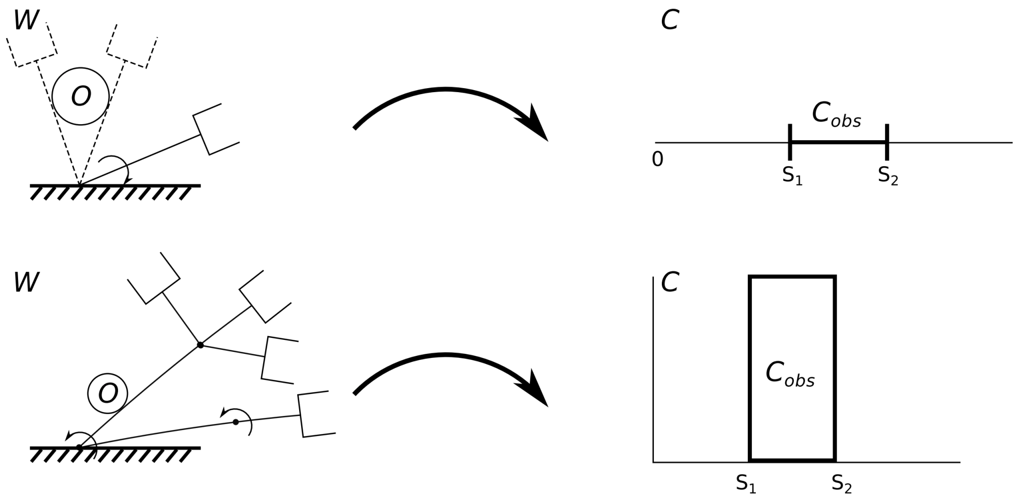




Workspace view hides the real difficulty — planning happens in configuration space.

What Is Configuration Space?

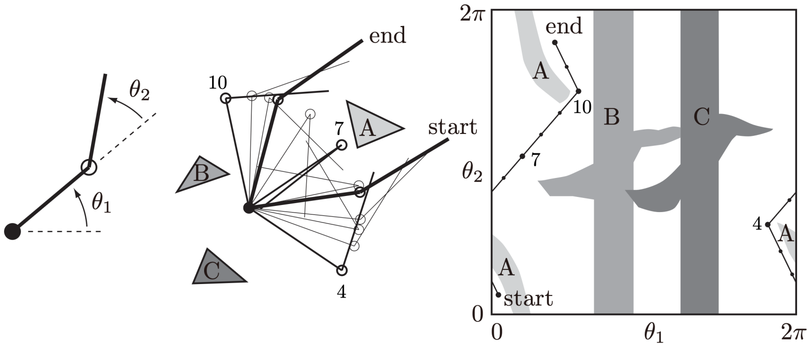
Configuration: the minimum set of parameters to specify every point on the robot.



Notation: In planning literature, ***q*** denotes configuration. In dynamics (Lectures 2–3), ***θ*** is common. Same concept, different convention.

Symbol	Meaning
$q \in \mathbb{R}^n$	Robot configuration (joint positions)
C	Configuration space — set of all possible q
n	Degrees of freedom

C-free and C-obstacle



Definitions

$$\mathcal{C}_{\text{obs}} = \{q \in \mathcal{C} \mid \mathcal{A}(q) \cap \mathcal{O} \neq \emptyset\}$$

$$\mathcal{C}_{\text{free}} = \mathcal{C} \setminus \mathcal{C}_{\text{obs}}$$

Symbol	Meaning
$A(q) \subset \mathbb{R}^3$	Robot body at configuration q
$O \subset \mathbb{R}^3$	Set of workspace obstacles

Key insight: C_obs has ***implicit geometry*** — you can test if a point is in collision (membership oracle) but cannot enumerate the boundary.

Learning Connection

The implicit geometry of C-obstacles — where you can test membership but cannot enumerate the boundary — is exactly where neural networks excel. SceneCollisionNet (2021) learns to predict collisions from point clouds, achieving 75× speedup. This paradigm recurs throughout robot learning: NeRFs, SDFs, and value functions.

Why Planning Is Hard

Computational Complexity

Motion planning is PSPACE-complete (Reif 1979, Canny 1988). Exact algorithms have doubly-exponential complexity in degrees of freedom.

Curse of Dimensionality

A 7-DOF arm lives in $\mathbb{R}^7$. Exact cell decomposition becomes intractable beyond ~3 DOF. Grid-based methods scale exponentially: $O(r^n)$ cells.

Narrow Passages

Even uniform random sampling can require exponentially many samples to find paths through tight corridors. P(sampling in narrow passage) → 0.

These three challenges motivate sampling-based approaches: don’t construct C-space — just probe it.

The Motion Planning Problem

Formal Statement

$$\text{Find } \tau : [0,1] \to \mathcal{C}_{\text{free}} \text{ such that } \tau(0) = q_{\text{start}},\ \tau(1) = q_{\text{goal}}$$

Symbol	Meaning
τ : [0,1] → C_free	Continuous collision-free path
q_start, q_goal	Start and goal configurations in C_free

Three Approaches to This Problem

Sampling-Based

Explore C_free by random sampling and incremental construction

Optimization-Based

Optimize a trajectory functional over a continuous representation

Learning-Based

Amortize planning across environments using neural networks

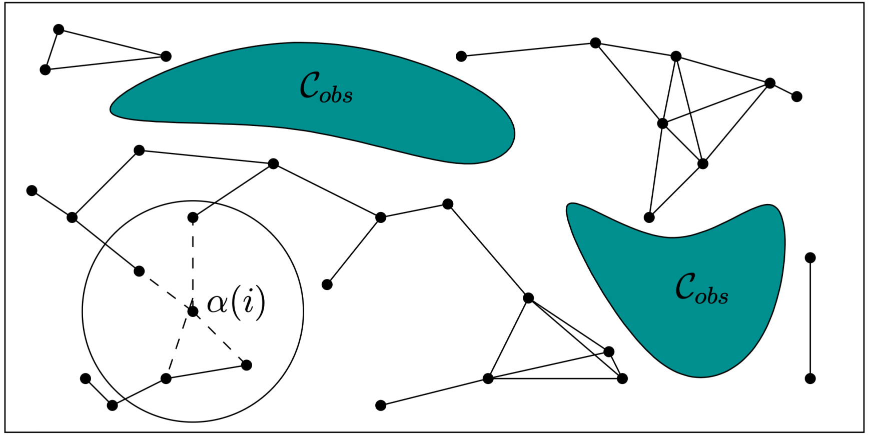
This lecture covers all three, plus the task-level planning that sits above them.

Sampling-Based Motion Planning

Finding paths by random exploration

The Sampling Idea

Core insight: Probe C_free by random sampling + collision checking, rather than constructing it explicitly.



Two Paradigms

PRM — Probabilistic Roadmap

Multi-query: build a roadmap once, reuse for many start/goal pairs.

RRT — Rapidly-Exploring Random Tree

Single-query: grow a tree from start toward goal. Best for one-shot planning.

Both are **probabilistically complete**: P(find path | path exists) → 1 as samples → ∞

Both require only a collision checker — no explicit C-obstacle construction needed.

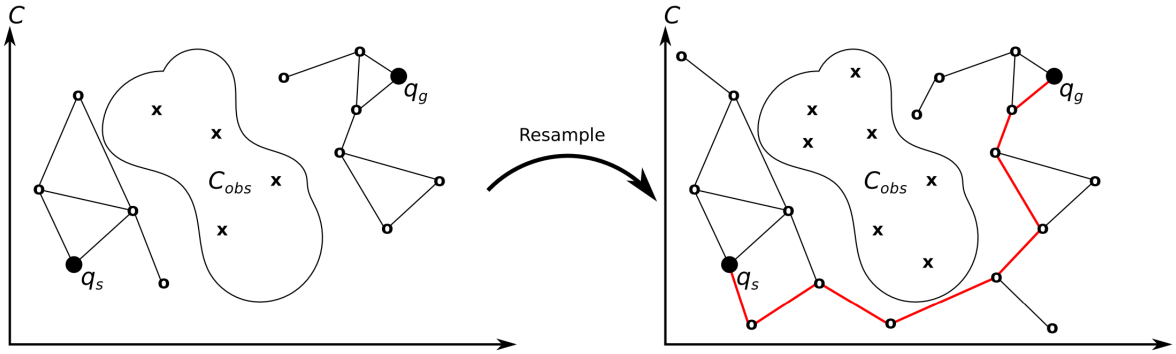
Probabilistic Roadmaps (PRM)

Phase 1: Learning (offline)

1. Sample N random configs in C_free
2. For each pair within radius r, attempt connection
3. Check edge for collisions (discretized)
4. Store valid connections as graph edges

Phase 2: Query (online)

1. Connect q_start and q_goal to nearby nodes
2. Run graph search (A*, Dijkstra)
3. Return path or report failure
4. Can reuse roadmap for new queries

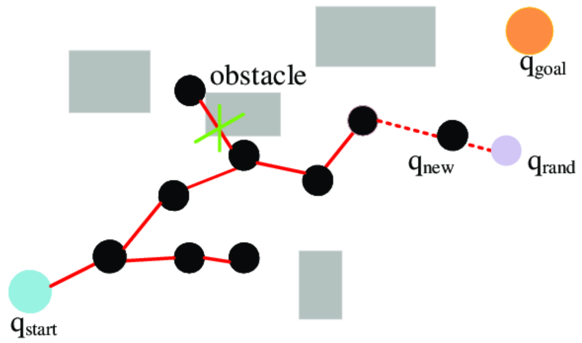


Kavraki, Švestka, Latombe & Overmars (1996)

Rapidly-Exploring Random Trees (RRT)

RRT Algorithm

```
Initialize T with q_start
for i = 1 to N:
    q_rand ← RandomSample(C)
    q_near ← NearestNode(T, q_rand)
    q_new ← Extend(q_near, q_rand, ε)
    if CollisionFree(q_near, q_new):
        AddVertex(T, q_new)
        AddEdge(T, q_near, q_new)
    if ‖q_new − q_goal‖ < δ:
        return ExtractPath(T)
```



Voronoi bias

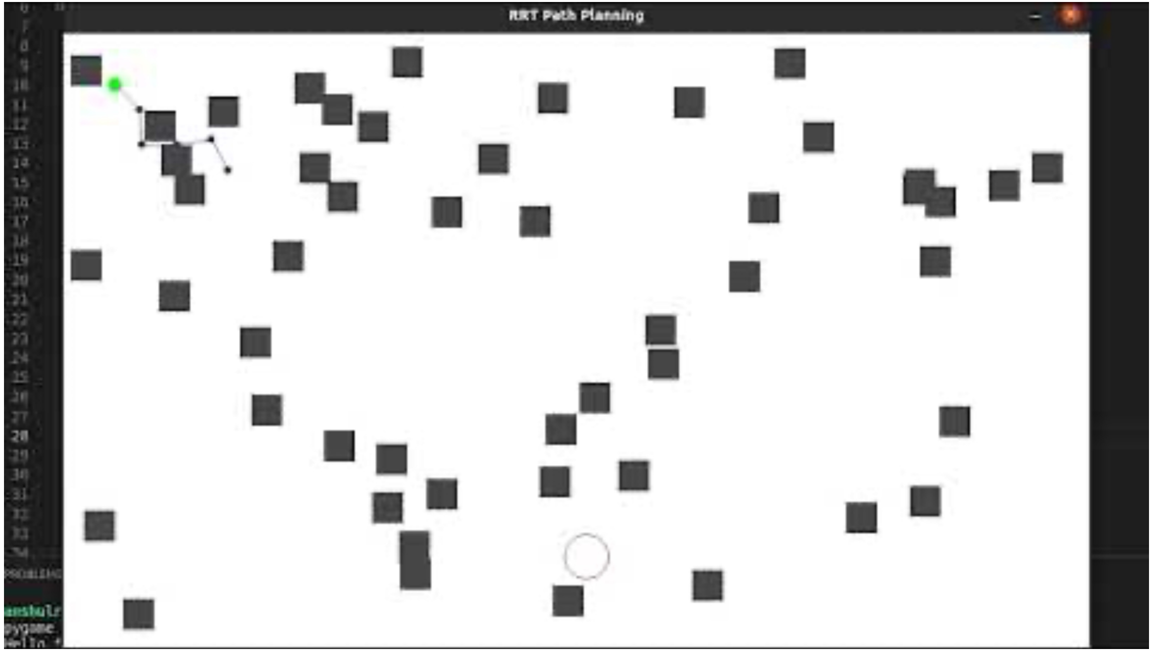
Nodes with larger Voronoi regions are more likely to be selected → rapid exploration of unexplored space.

Extension formula:

$$q_{\text{new}} = q_{\text{near}} + \frac{\varepsilon}{\|q_{\text{rand}} - q_{\text{near}}\|}(q_{\text{rand}} - q_{\text{near}})$$

LaValle (1998); Kuffner & LaValle (2000)

RRT in Action



Observations

Path found

Collision-free, connects start to goal

Path quality

Jerky, suboptimal — piecewise-linear segments

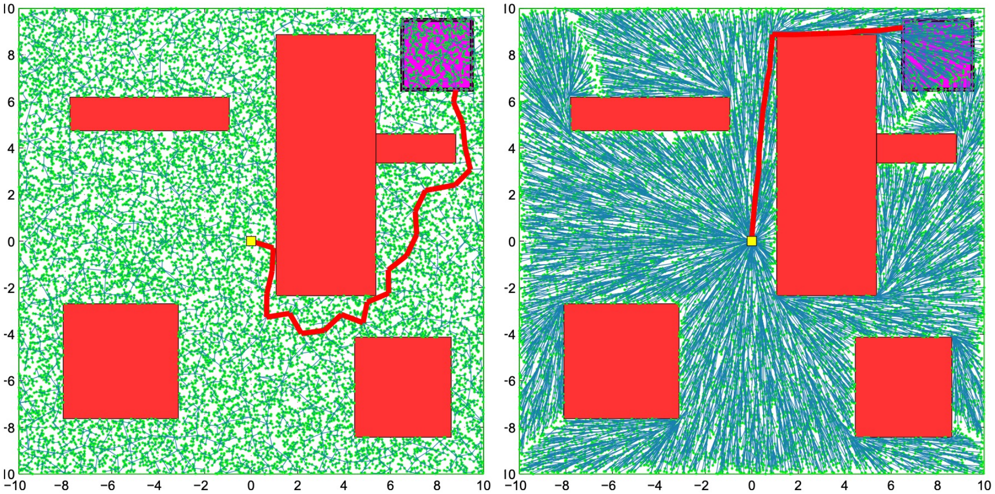
Missing

Smoothness, dynamic feasibility, timing

The Optimality Gap

Karaman & Frazzoli (2011):

As n → ∞, the probability that RRT converges to the optimal path → 0. This is not an implementation bug — it is a fundamental property of the algorithm.



RRT path vs. optimal path on same 2D problem

Why?

RRT commits to the first connection for each node. Once added, the parent edge is permanent — no mechanism to rewire when better paths are found. The tree locks in early suboptimal decisions.

This motivates RRT*: add a rewiring step that corrects suboptimal connections.

RRT*: Asymptotic Optimality

Two modifications to RRT that guarantee asymptotic optimality:

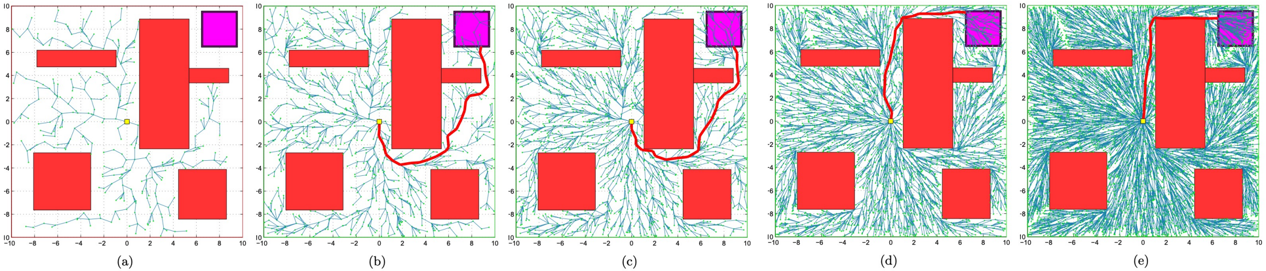
1. Near-Neighbor Search

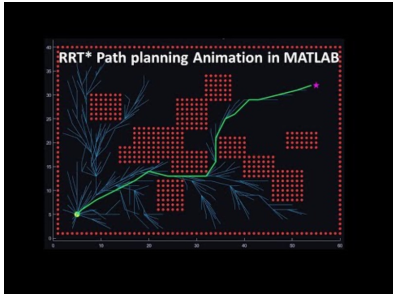
Instead of connecting only to nearest node, find all nodes within radius r(n). Choose the one that

$$r(n) = \gamma \left(\frac{\log n}{n} \right)^{1/d}$$

2. Rewiring

After adding q_new, check if routing through q_new reduces the cost to any nearby node. If so, rewire their parent edge. This corrects the suboptimal decisions that plague RRT.


(a) (b) (c) (d) (e)



Variables: n = number of samples, d = C-space dimension, γ = constant depending on C-free volume

Cost: only constant-factor overhead per iteration — same O(n log n) as RRT.

Informed RRT* and BIT*

Accelerating convergence by focusing sampling where improvement is possible:

Informed RRT*

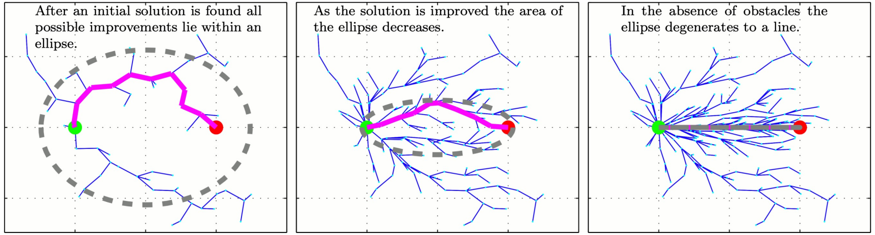
Once an initial solution c_best is found, sample only within the prolate hyperspheroid of configurations that could improve it:

$$\{q \in \mathcal{C} \mid \|q - q_{\text{start}}\| + \|q - q_{\text{goal}}\| \le c_{\text{best}}\}$$

As c_best decreases, the ellipsoid shrinks → focused sampling.

BIT* (Batch Informed Trees)

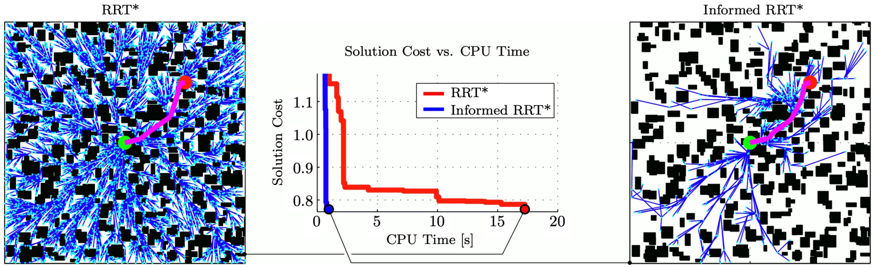
Unifies graph search and sampling-based planning. Processes samples in batches over implicit random geometric graphs with heuristic-guided search (like A*). Each batch refines the solution. Combines the best of RRT* (any-time) and PRM* (graph search).


After an initial solution is found all possible improvements lie within an ellipse.
As the solution is improved the area of the ellipse decreases.
In the absence of obstacles the ellipse degenerates to a line.

59 iterations, $c_{\text{best}} = 148.24$

175 iterations, $c_{\text{best}} = 107.12$

1142 iterations, $c_{\text{best}} = 100$



17.3 seconds, $c_{\text{best}} = 0.77$

0.9 seconds, $c_{\text{best}} = 0.77$

Gammell, Srinivasa & Barfoot (2014, 2015)

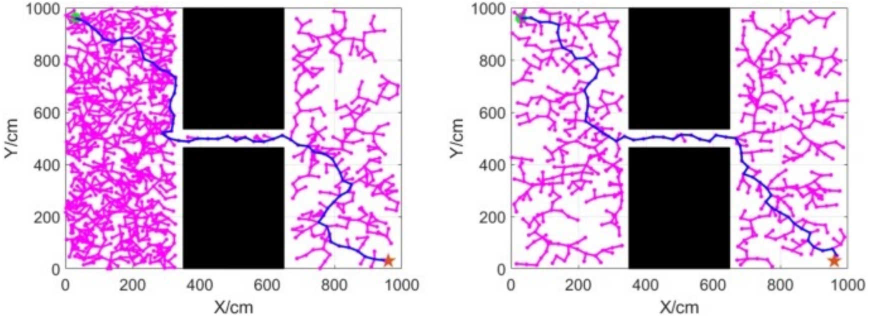
Comparing the Variants

Algorithm	First Path	Optimality	Convergence	Best For
RRT	Fast	None	N/A	Quick feasibility check
RRT*	Same as RRT	Asymptotic	Slow (uniform)	Provably optimal paths
Informed RRT*	Same as RRT	Asymptotic	Fast (focused)	Rapid convergence to optimal
BIT*	Moderate	Asymptotic	Fastest	Best anytime performance

Key Insight: Feasible-First, Then Optimize

The RRT → RRT* → Informed RRT* progression illustrates a general pattern: start with any feasible solution, then iteratively improve within a shrinking feasible set. This pattern recurs in trajectory optimization (initial guess → gradient refinement), RL (initial policy → policy improvement), and VLA fine-tuning (pre-trained backbone → task adaptation).

Limitations of Classical Sampling



（a） RRT

(b) RRT*

Core Challenges

Narrow passages: Probability of sampling in tight corridors → 0. Exponential samples needed.

High dimensions: 7+ DOF: volume of C_free vanishes relative to C. Collision checking dominates.

Constrained spaces: Manifold constraints (e.g., holding a cup level) make sampling harder.

Runtime bottleneck: Collision checking: 70–90% of total planning time in practice.

These limitations motivate two responses: optimization-based planning and learning-based acceleration.

Where does the time go?

Collision checking: 70–90% of runtime. Forward kinematics → geometry queries → GJK/EPA per link per waypoint. This is embarrassingly parallel but still expensive — and it’s the natural target for neural acceleration.

Neural Motion Planning

Key idea: replace uniform sampling with learned, task-conditioned distributions.

MPNet (2019)

Encoder + planning network

Learned path generation

< 1s planning time

Hybrid: neural + classical fallback

Probabilistic completeness preserved

MπNets (2023)

Reactive policy from depth images

Trained on 3M+ problems

46% improvement over baselines

Handles dynamic obstacles

Real-time replanning

Neural MP (2024)

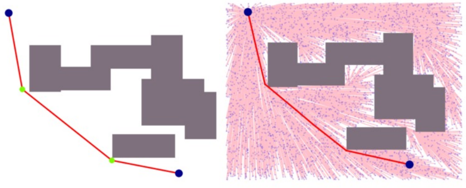
Generalist neural motion planner

64 real-world tasks validated

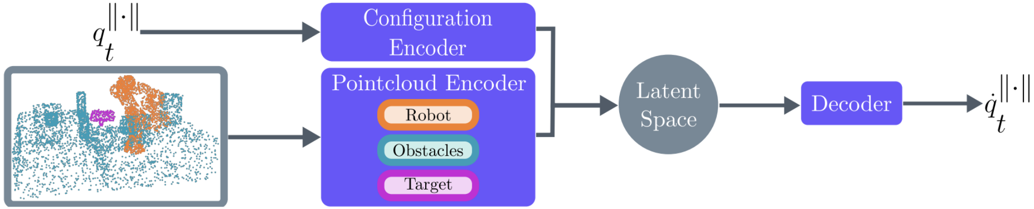
23% improvement over sampling

Point cloud + language input

Foundation-model approach



(a) MPNet (b) RRT*



> **Learning Connection: Amortized Inference**
>
> Neural motion planners amortize search across a training distribution — the same idea behind VAEs and diffusion models. Trade-off: millisecond planning but dependence on the training distribution.

Learned Collision Checking

Neural networks accelerate the planning bottleneck while preserving formal guarantees.

SceneCollisionNet (2021)

Neural collision prediction from point clouds. Replaces geometric GJK/EPA queries with a single forward pass. 75× speedup over traditional methods. Enables real-time rearrangement planning. Used as a drop-in replacement for collision checking in any sampling-based planner.

Neural Informed RRT* (2024)

PointNet++ learns sampling distributions that bias RRT* toward promising regions. Key contribution: preserves asymptotic optimality guarantees despite learned sampling. Faster convergence without sacrificing theoretical properties. Bridges the learning–classical divide.

Performance Summary

System	Approach	Speedup	Guarantees
SceneCollisionNet	Learned collision checking	75×	Drop-in replacement
Neural Informed RRT*	Learned sampling bias	~3–5× faster convergence	Asymptotic optimality

The Hybrid Approach

The most effective systems combine learned components (speed) with classical algorithms (guarantees). This is a recurring theme: use learning where it helps, keep formal properties where they matter.

Trajectory Optimization

From finding paths to finding good trajectories

From Paths to Trajectories

Sampling planners produce collision-free paths — but real robots need more:

Path (from RRT/PRM)

Piecewise-linear waypoints

Collision-free only

No timing information

Discontinuous velocities

Jerky, not executable as-is

Trajectory (what robots need)

Smooth, continuous path

Collision-free + dynamically feasible

Time-parameterized: q(t)

Bounded velocities & accelerations

Directly executable by controllers

Trajectory optimization directly optimizes over a continuous representation to bridge this gap.

The Trajectory Optimization Problem

General Formulation

$$\min_{\xi} \sum_{t=1}^{N} c_{\mathrm{obs}}(\xi_t) + \lambda \cdot c_{\mathrm{smooth}}(\xi) \quad \text{s.t. } \xi_1 = q_{\mathrm{start}}, \; \xi_N = q_{\mathrm{goal}}$$

Symbol	Meaning
ξ ∈ ℝ^{N×d}	Discretized trajectory (N waypoints in d-dim C-space)
c_obs(q), c_smooth(ξ)	Obstacle cost and smoothness cost (e.g., ½ ξ^{T} K ξ)
λ ∈ ℝ$^{+}$, N ∈ ℕ	Trade-off weight; number of waypoints

Three major algorithms all solve variants of this formulation:

CHOMP
Functional gradient descent with covariant smoothing

STOMP
Stochastic sampling, no gradients needed

TrajOpt
Sequential convex optimization with continuous collision

All require an initial guess — quality of initialization strongly affects convergence.

CHOMP: Functional Gradient Descent

Zucker et al. (2013), IJRR

$$\xi \leftarrow \xi - \frac{1}{\eta} K^{-1} \bar{\nabla} J(\xi)$$

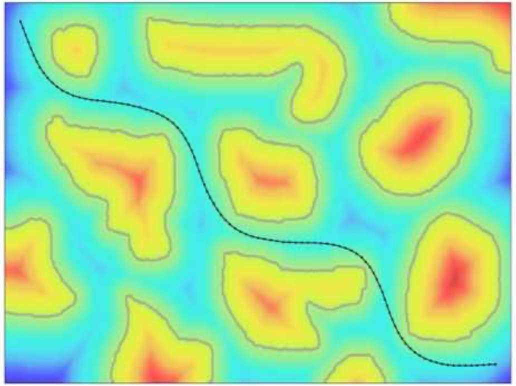
$$c_{\text{smooth}}(\xi) = \frac{1}{2} \xi^\top K \xi$$

$$c_{\text{obs}}(q) = \sum_{x \in \text{body}(q)} \max(\epsilon - d(x), 0)$$

Variables: K = finite-difference precision matrix (encodes smoothness); η = step size; d(x) = signed distance to nearest obstacle; ε = safety margin. CHOMP requires differentiable cost — signed distance fields provide this.

Why K^{-1} matters

The K^{-1} preconditioner produces smooth deformations rather than jagged corrections. Standard gradient descent would update each waypoint independently; covariant descent updates the entire trajectory coherently.



STOMP and TrajOpt

STOMP (Stochastic)

Generate noisy trajectory rollouts around current solution. Evaluate costs (no gradients needed). Weight and combine: low-cost rollouts contribute more. Can escape local minima through stochastic exploration. Works with non-differentiable cost functions.

TrajOpt (Sequential Convex)

Approximate non-convex problem as sequence of convex subproblems. Continuous-time collision checking via signed distance. Penalty method for constraint satisfaction. Scales to high-DOF: demonstrated on 34-DOF humanoid. Fast convergence in practice.

Method Comparison

	CHOMP	STOMP	TrajOpt
Approach	Functional gradient	Stochastic sampling	Sequential convex
Gradients?	Required	Not required	Required
Local minima	Susceptible	Can escape	Susceptible
Scalability	Moderate DOF	Moderate DOF	High DOF (34+)
Key strength	Smooth updates	Gradient-free	Continuous collision

Kalakrishnan et al. (2011); Schulman et al. (2014)

Trajectory Optimization in Practice

When to Use Each Method

CHOMP: Differentiable costs available, moderate DOF, smooth solutions needed

STOMP: Non-differentiable costs, need to escape local minima, complex cost landscapes

TrajOpt: High-DOF systems, tight constraints, continuous collision needed

Common Pitfalls

Local minima: All methods can get stuck. CHOMP/TrajOpt are deterministic — same init → same result. Multiple restarts or STOMP help.

Initialization sensitivity: Quality of initial guess dominates solution quality. Straight-line init fails in cluttered scenes.

Penalty vs. constraint: Soft penalties may violate constraints; hard constraints may be infeasible. Tuning λ is an art.

The Initialization Problem

The dependence on good initial guesses directly motivates hybrid approaches: use learning to generate good seeds, then refine with classical optimization. This is exactly what DiffusionSeeder does (next slide).

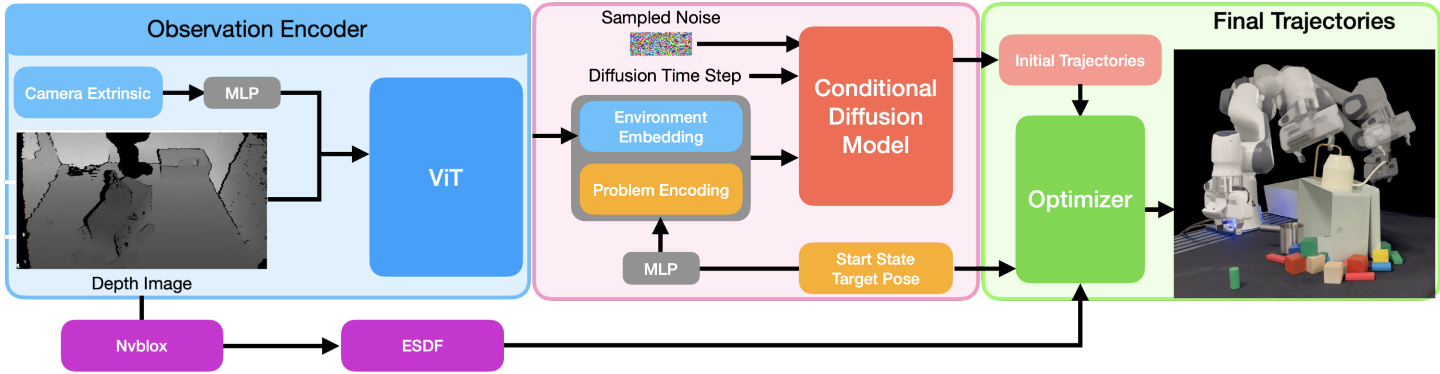
Diffusion Models Meet Trajectory Optimization

DiffusionSeeder (2024)

Diffusion model generates diverse seed trajectories from point cloud input. cuRobo (GPU-accelerated optimizer) refines seeds in parallel. Result: 12× speedup over baselines, 86% success rate in 26ms. Multiple seeds → parallel refinement → best solution.

SafeDiffuser (2025)

Integrates control barrier functions (CBFs) directly into the denoising process. Each denoising step projects onto the safe set. Provides formal safety guarantees during generation — not just post-hoc checking. Safety by construction, not filtering.



Learning Connection: Learned Init + Classical Refinement

DiffusionSeeder exemplifies a recurring pattern: learning handles the hard global component (generating good initializations from raw perception) while classical optimization provides local refinement with feasibility guarantees.

Sampling vs. Optimization vs. Learning

Sampling-Based

RRT, RRT, PRM, BIT**

Completeness
Probabilistic

Computation
Seconds to minutes

Initialization
Not required

Generalization
Any C-space

Optimization-Based

CHOMP, STOMP, TrajOpt

Completeness
Local only

Computation
Sub-second

Initialization
Critical

Generalization
Any C-space

Learning-Based

MPNet, MπNets, Neural MP

Completeness
Training-dependent

Computation
Milliseconds

Initialization
Implicit (learned)

Generalization
Distribution-limited

Current trend: hybrid approaches that combine sampling (exploration) + optimization (refinement) + learning (speed).

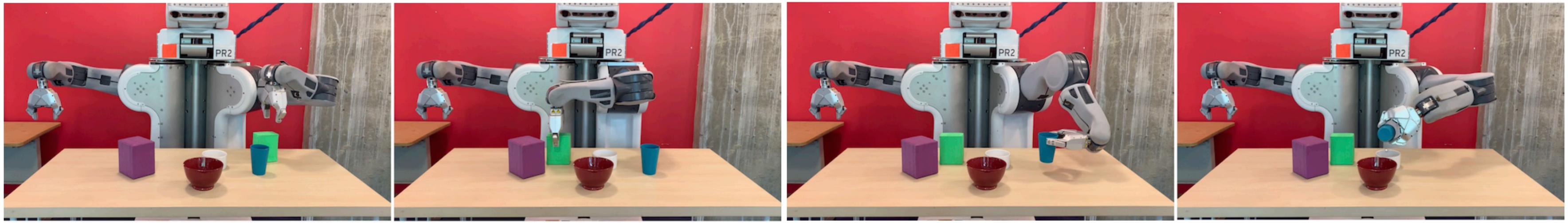
Task and Motion Planning

When the robot must reason about what to do, not just how to move

Beyond Single Motions

Motion planning finds a path from A to B. But real tasks require multi-step reasoning:

> **Example: Set the dinner table.** Requires: sequencing pick-and-place actions, reasoning about spatial relationships (plate before glass), ensuring each action is geometrically feasible (can the arm reach?), handling object rearrangement when items are in the way.



The specified goal is for the contents of the blue cup to end up in the white bowl.

What motion planning alone cannot do:

Decide what to do
Which actions, in what order?

Reason about effects
How does each action change the world?

Handle dependencies
Some actions require prior rearrangement.

The TAMP Problem

Formal Structure

$$\text{Find } (\pi, \theta_{1:K}) \text{ s.t. } \forall k : \mathrm{Pre}(a_k) \subseteq s_k,\ s_{k+1} = (s_k \setminus \mathrm{Eff}^-(a_k)) \cup \mathrm{Eff}^+(a_k),\ \theta_k \text{ feasible}$$

Symbol	Meaning
s ∈ S	Symbolic state (set of ground predicates, On(plate, counter), HandEmpty, Clear(table))
π = (a$_1$,...,a_K)	Task plan (sequence of actions) — not a policy, Pick(plate), Place(plate, table), Pick(glass)
θ_k	Continuous parameters for action k (grasps, placements, paths)
Pre(a), Eff±(a), K	Preconditions, add/delete effects; number of actions

Two coupled layers:

Symbolic (task) layer: What actions to take, in what order. Discrete search over action sequences using preconditions and effects.

Geometric (motion) layer: How to execute each action. Continuous optimization over grasps, placements, and collision-free motions.

Notation: π here denotes a task plan (action sequence), not a policy.

The Interface Problem

Why TAMP is fundamentally hard: bidirectional coupling between layers.

Symbolic → Geometric: The task plan constrains which motions are needed. You can't plan a grasp until you know what to grasp.

Geometric → Symbolic: Geometric infeasibility forces plan revision. A blocked shelf means the robot must first move obstructors — changing the entire plan.

Key Insight: The Discrete–Continuous Interface

The interface problem — combining discrete reasoning (logic, language) with continuous optimization (geometry, physics) — is a specific instance of a fundamental AI challenge. The same tension appears in program synthesis, scientific discovery, and LLM-based robotics. Understanding TAMP provides a framework for recognizing this pattern.

Three Classical TAMP Paradigms

Hierarchical

Kaelbling & Lozano-Pérez, 2011

Plan top-down, refine lazily. Generate abstract plan first, then attempt geometric realization. Backtrack if geometry fails. Fast when most plans are feasible.

Optimization-Based

Toussaint, 2015 (LGP)

Joint nonlinear program over task and motion variables simultaneously. Logic-Geometric Programming (LGP): symbolic skeleton + continuous optimization. Elegant but computationally expensive.

Constraint-Based

Garrett et al., 2020 (PDDLStream)

Extend PDDL with sampling streams that generate geometric parameters on demand. Fast Downward handles symbolic search. Adaptive: interleaves planning and sampling.

	Hierarchical	Optimization	Constraint
Integration	Sequential	Joint	Interleaved
Completeness	Resolution	Local	Probabilistic
Scalability	Large tasks	Small tasks	Medium tasks

PDDLStream: A Closer Look

Key idea: extend PDDL with streams — procedures that lazily generate geometric parameters.

Step 1: Symbolic Search

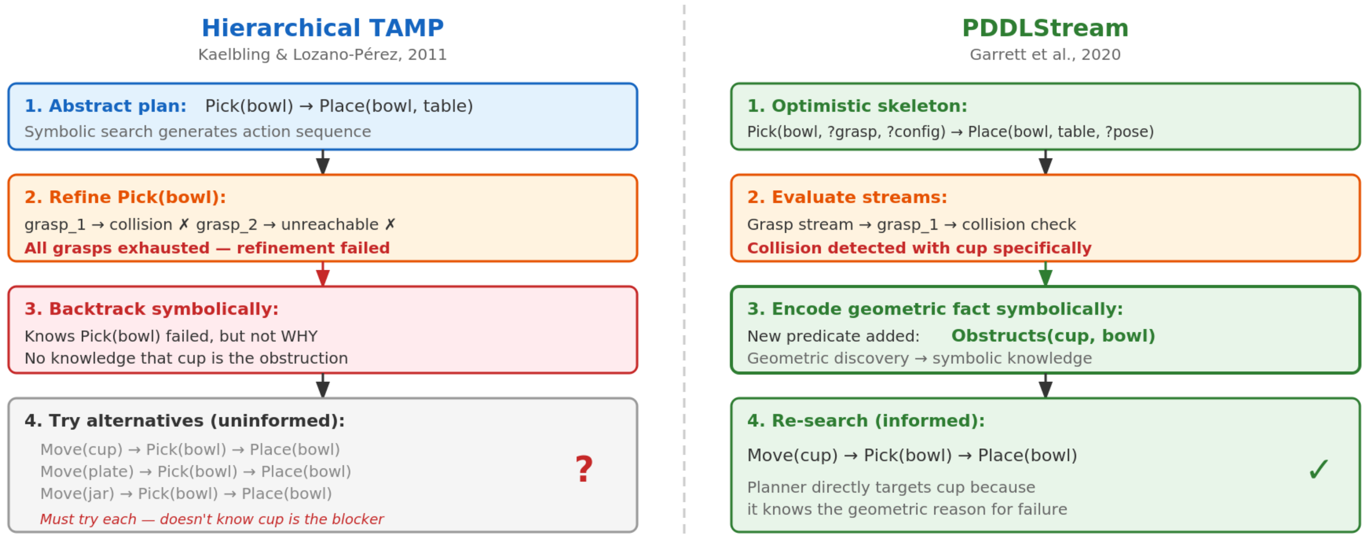
Fast Downward planner generates a skeleton plan using PDDL predicates. Stream outputs treated as optimistic free variables.

Step 2: Stream Evaluation

Streams sample concrete values: IK(pose) → config, MotionPlan(start, goal) → trajectory, Grasp(obj) → grasp pose.

Step 3: Geometric Validation

Check if sampled parameters satisfy all geometric constraints. If not, inform planner and re-search with updated knowledge.



Hierarchical TAMP

Kaelbling & Lozano-Pérez, 2011

1. Abstract plan: Pick(bowl) → Place(bowl, table)

Symbolic search generates action sequence

2. Refine Pick(bowl):

grasp_1 → collision ✗ grasp_2 → unreachable ✗

All grasps exhausted — refinement failed

3. Backtrack symbolically:

Knows Pick(bowl) failed, but not WHY

No knowledge that cup is the obstruction

4. Try alternatives (uninformed):

Move(cup) → Pick(bowl) → Place(bowl)

Move(plate) → Pick(bowl) → Place(bowl)

Move(jar) → Pick(bowl) → Place(bowl)

?

Must try each — doesn't know cup is the blocker

PDDLStream

Garrett et al., 2020

1. Optimistic skeleton:

Pick(bowl, ?grasp, ?config) → Place(bowl, table, ?pose)

2. Evaluate streams:

Grasp stream → grasp_1 → collision check

Collision detected with cup specifically

3. Encode geometric fact symbolically:

New predicate added: **Obstructs(cup, bowl)**

Geometric discovery → symbolic knowledge

4. Re-search (informed):

Move(cup) → Pick(bowl) → Place(bowl)

✓

Planner directly targets cup because

it knows the geometric reason for failure

Where Classical TAMP Struggles

Four fundamental limitations that motivate the shift to foundation models:

1. Domain Engineering: Manually specifying predicates, actions, preconditions, and effects for every new domain. A table-setting domain requires different symbols than a cooking domain. Months of expert effort per domain.

2. Open-World Reasoning: Classical TAMP operates in a closed world: every relevant object, predicate, and action must be pre-defined. Cannot handle novel objects (“what is this tool?”) or unexpected situations.

3. Scalability: Combinatorial explosion in long-horizon tasks. A 10-step plan with 5 objects and 10 grasps per object: search space exceeds 10^{10}. Heuristics help but don’t eliminate the issue.

4. Brittleness: Rigid symbolic representations break on partial observability, sensor noise, or execution errors. No natural mechanism for recovery or replanning from unexpected states.

Each limitation maps directly to a foundation model capability →

The Promise of Foundation Models

TAMP Limitation		Foundation Model Response
Domain Engineering	→	LLMs provide common-sense knowledge and action semantics without manual specification
Open-World Reasoning	→	VLMs handle open vocabulary, novel objects, and unstructured environments
Scalability	→	Learned policies amortize planning cost across training distribution
Brittleness	→	Language enables flexible recovery, replanning, and human-in-the-loop correction

Open Challenges Remain

Grounding, verification, and long-horizon consistency are still unsolved. Foundation models provide powerful priors but not guarantees.

Foundation Models for Planning

From hand-crafted pipelines to learned reasoning

Three Paradigms for FM-Based Planning

From most modular to most integrated:

Paradigm 1

LLM as Task Planner

SayCan, Inner Monologue, ProgPrompt

LLM selects from a fixed skill library. Affordance models ground in physics. Most modular and interpretable. Limited by predefined skill set.

Separate reasoning from acting

Paradigm 2

LLM as Code Generator

Code as Policies, VoxPoser, Eureka

LLM writes executable code or reward functions. Spatial reasoning via API composition. Qualitative expansion in expressiveness. Requires well-designed APIs.

Programs > skill sequences

Paradigm 3

End-to-End VLA

RT-2, OpenVLA, π_0

Vision-Language-Action model maps directly from perception + instruction to actions. No explicit planning or skills. Most flexible but least interpretable.

Implicit planning via representations

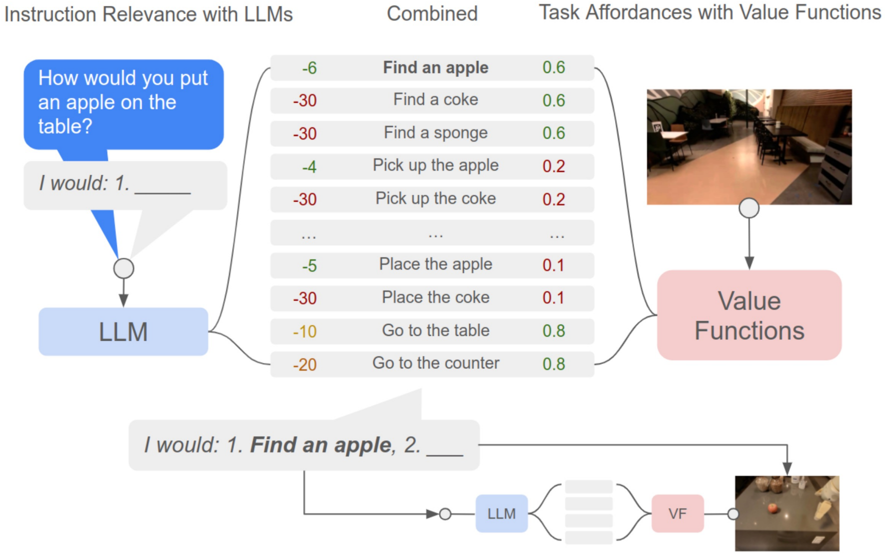
Each paradigm offers a different answer to the grounding problem.

Paradigm 1: LLM as Task Planner

SayCan (2022): factored scoring separates semantic reasoning from physical grounding.

Variables: i = instruction, a_t = candidate action, ℓ_t = language context, s_t = current state, $p(i \mid a_t, \ell_t)$ = "Say" score (LLM), $p(\text{success} \mid a_t, s_t)$ = "Can" score (affordance)

$$\pi^*(a_t \mid i, s_t) = \arg\max_{a_t} \; p(i \mid a_t, \ell_t) \cdot p(\text{success} \mid a_t, s_t)$$



Instruction Relevance with LLMs
Combined
Task Affordances with Value Functions
How would you put an apple on the table?
I would: 1. _____
LLM
Value Functions
I would: 1. Find an apple, 2. ___
LLM
VF

-6	**Find an apple**	0.6
-30	Find a coke	0.6
-30	Find a sponge	0.6
-4	Pick up the apple	0.2
-30	Pick up the coke	0.2
…	…	…
-5	Place the apple	0.1
-30	Place the coke	0.1
-10	Go to the table	0.8
-20	Go to the counter	0.8

"Say" score (LLM): How useful is this action for the instruction? LLM rates each candidate skill based on semantic relevance to the task.

"Can" score (Affordance): Can the robot actually do this right now? Learned value function estimates success probability given current state.

Result: 84% planning success on 101 real kitchen tasks. Limitation: open-loop — no feedback during execution.

Closing the Loop

SayCan plans open-loop. Two systems address this with feedback:

Inner Monologue (2022)

Closed-loop LLM planning with three feedback sources:

- Success detection: "Did the action succeed?"
- Scene description: VLM describes current state
- Human correction: "No, I meant the other cup"

LLM integrates feedback and replans. Converts open-loop to closed-loop without retraining.

ProgPrompt (2023)

Python-like prompts with assertion-based precondition checks:

```
assert is_reachable(cup)
assert is_graspable(cup)
grasp(cup)
```

If assertion fails, LLM generates recovery plan. Bridges natural language and programmatic verification.

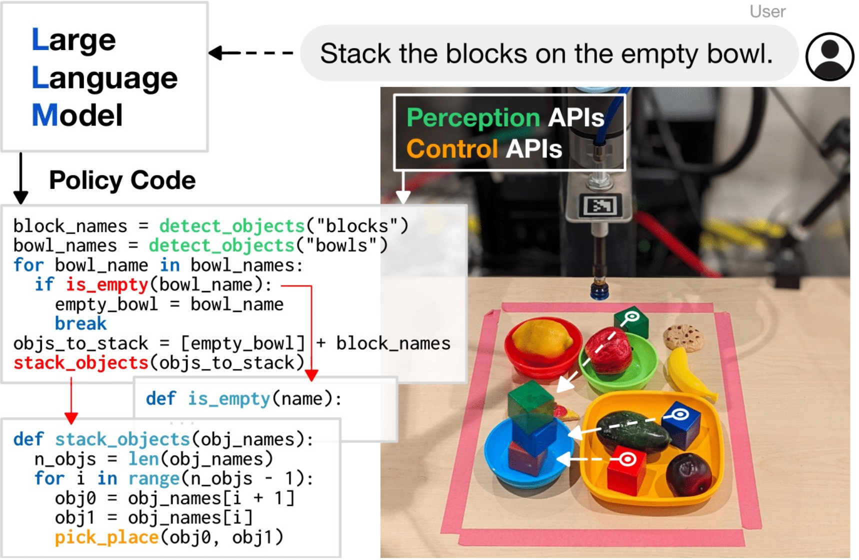
From Skills to Programs

Both systems extend the LLM-as-planner paradigm but remain limited to predefined skills. Paradigm 2 breaks this constraint: instead of selecting from skills, the LLM writes the program itself.

Paradigm 2: LLM as Code Generator

Code as Policies (2023): LLM generates executable Python with spatial reasoning and control flow.

> **Key shift:** Space of all programs >> space of all skill sequences. The LLM can compose spatial primitives, loops, conditionals, and API calls into behaviors never explicitly programmed. Language instructions become code that interfaces directly with perception and control APIs.



Stack the blocks on the empty bowl.

```
block_names = detect_objects("blocks")
bowl_names = detect_objects("bowls")
for bowl_name in bowl_names:
  if is_empty(bowl_name):
    empty_bowl = bowl_name
    break
objs_to_stack = [empty_bowl] + block_names
stack_objects(objs_to_stack)
```

```
def is_empty(name):
  ...
```

```
def stack_objects(obj_names):
  n_objs = len(obj_names)
  for i in range(n_objs - 1):
    obj0 = obj_names[i + 1]
    obj1 = obj_names[i]
    pick_place(obj0, obj1)
```

What code enables:

Spatial reasoning

get_pos(), compute_centroid(), sort_by_distance()

Control flow

for obj in objects, if is_heavy(obj), while not aligned()

API composition

Chain perception, planning, and control primitives arbitrarily

From Code to 3D Value Maps

VoxPoser (2023): LLM generates code that composes 3D affordance and constraint maps in voxel space.

Language Instruction

"Open the drawer and put the apple inside"

→

LLM → Code

Generates Python composing 3D value maps from perception APIs

→

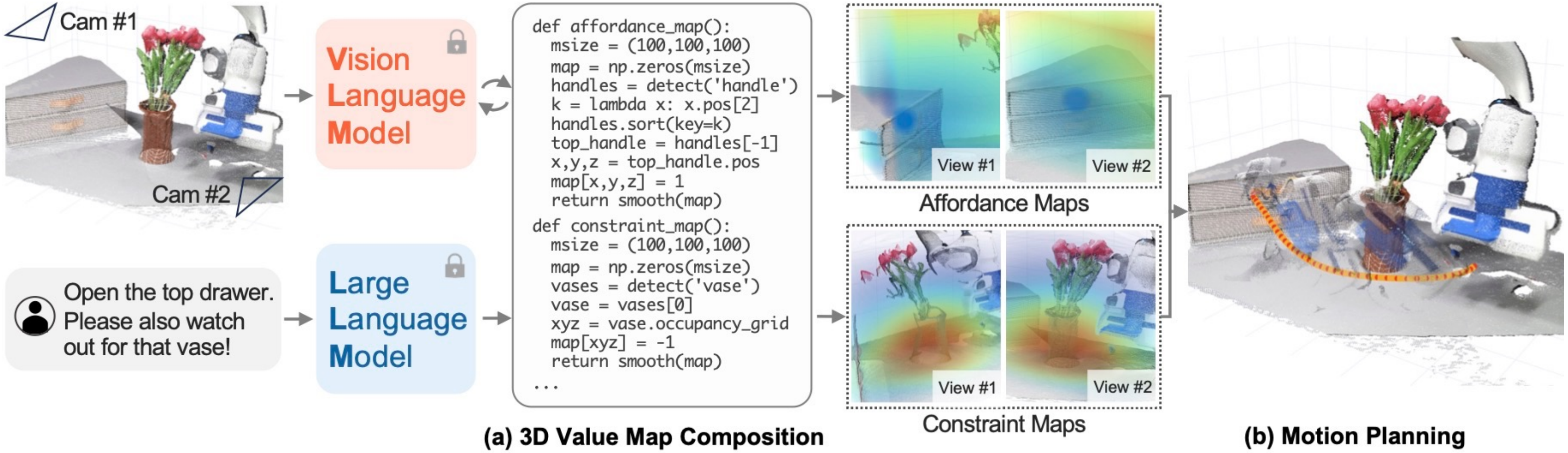
3D Voxel Maps

Affordance (attract) + constraint (repel) maps in workspace

→

MPC Execution

Model-predictive control synthesizes trajectory zero-shot



```
def affordance_map():
  msize = (100,100,100)
  map = np.zeros(msize)
  handles = detect('handle')
  k = lambda x: x.pos[2]
  handles.sort(key=k)
  top_handle = handles[-1]
  x,y,z = top_handle.pos
  map[x,y,z] = 1
  return smooth(map)
def constraint_map():
  msize = (100,100,100)
  map = np.zeros(msize)
  vases = detect('vase')
  vase = vases[0]
  xyz = vase.occupancy_grid
  map[xyz] = -1
  return smooth(map)
...
```

(a) 3D Value Map Composition

(b) Motion Planning

Grounding Through Spatial Structure

VoxPoser shows that code generation can go beyond sequencing skills: the LLM composes 3D spatial representations that directly interface with low-level control. The “code” is a specification of where to go, not what to do.

LLMs as Reward Engineers

Instead of generating plans or code, the LLM designs the objective function:

Eureka (2024)

GPT-4 writes and iteratively improves reward functions for reinforcement learning.

Pipeline: environment code → GPT-4 generates reward → train RL policy → evaluate → GPT-4 refines based on training curves.

83% surpass human expert rewards on 29 diverse tasks. Enables pen spinning on a simulated Shadow Hand.

Language to Rewards (2023)

Natural language instructions → reward code → MuJoCo MPC optimization.

Two-stage: LLM first generates a "reward sketch" (structured description), then translates to executable reward function.

90% success on 17 manipulation and locomotion tasks. No RL training — direct optimization.

The Pattern: LLM Designs, Classical Algorithm Executes

The LLM provides the what (reward specification); RL or MPC provides the how (policy optimization). Neither alone would succeed — the LLM lacks physics, the optimizer lacks semantics. Together: superhuman reward design.

Paradigm 3: End-to-End VLA Models

RT-2 (2023)

55B VLM + robot data

Actions as text tokens

Emergent reasoning abilities

Transfers web knowledge

Google DeepMind

OpenVLA (2024)

7B open-source VLA

Outperforms RT-2-X by 16.5%

Fine-tunable on custom data

970K robot episodes

Democratizes VLA research

π_0 (2024)

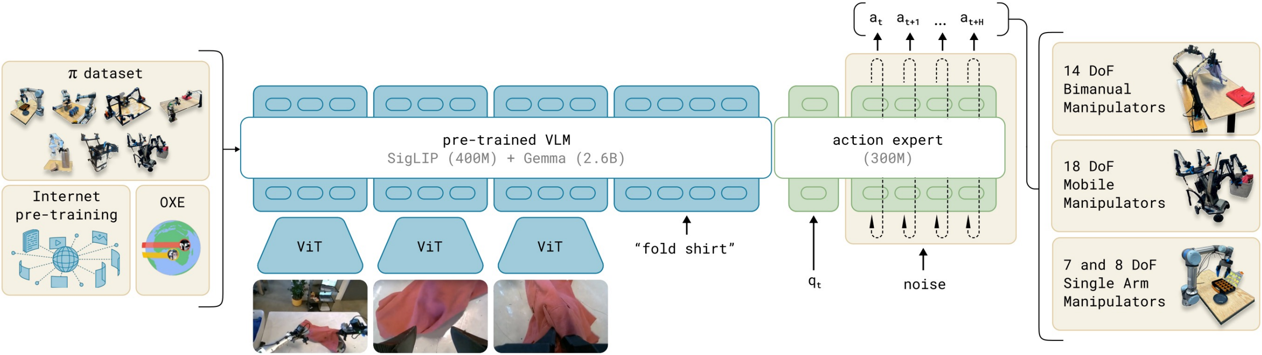
Flow matching architecture

50 Hz action generation

8 embodiments, dexterous tasks

Laundry folding, table bussing

Physical Intelligence


π dataset
Internet pre-training
OXE
pre-trained VLM
SigLIP (400M) + Gemma (2.6B)
ViT
ViT
ViT
"fold shirt"
action expert
(300M)
a_t a_{t+1} ... a_{t+H}
q_t
noise
14 DoF Bimanual Manipulators
18 DoF Mobile Manipulators
7 and 8 DoF Single Arm Manipulators

Learning Connection: Implicit Planning

VLA models represent the logical endpoint of this lecture's trajectory: from explicit planning (RRT, trajectory optimization) through structured integration (TAMP) to fully implicit planning via learned representations. Whether to embrace this — sacrificing modularity and formal guarantees for flexibility — is one of the central debates in embodied AI today.

The Paradigm Shift

What VLA models change — the entire TAMP pipeline absorbed into one network:

Trade-offs

What we gain

No explicit planning or hand-crafted skills

No domain engineering or predicate specification

Direct perception-to-action mapping

Flexible, generalizable behaviors

What we lose

Interpretability and debuggability

Formal guarantees (completeness, optimality)

Massive data requirements (100K+ demonstrations)

Safety verification becomes much harder

The field is actively exploring how to get the best of both worlds.

The Grounding Problem

How each paradigm grounds language in physics — the central challenge:

Paradigm	System	Grounding Mechanism	Limitation
1: Task Planner	SayCan	Learned affordance values	Fixed skill set
2: Code Gen	Code as Policies	API constraints + perception	API design burden
2: Code Gen	VoxPoser	3D spatial value maps	Voxel resolution limits
3: End-to-End	VLA (RT-2, π_0)	Training data distribution	Out-of-distribution failures

No Paradigm Fully Solves Grounding

None of these approaches guarantee grounding in novel environments or safety-critical applications. This remains the central open problem connecting planning, perception, and action — and a major theme of this course.

Open Challenges

Four frontier challenges for FM-based planning and manipulation:

1. Long-Horizon Consistency: Current FMs struggle with plans beyond 5–10 steps. Error accumulates; LLMs lose track of state over long sequences. Active research: hierarchical prompting, world models, memory mechanisms.

2. Physical Verification & Safety: FMs hallucinate plans that violate physics. No formal guarantees on collision avoidance or constraint satisfaction. Active research: verifiers, control barrier functions, runtime monitors.

3. Compositional Generalization: Can a model trained on "pick red cup" and "place on shelf" generalize to "pick red cup and place on shelf"? Systematic compositionality remains elusive. Active research: neuro-symbolic methods, program synthesis.

4. Sample Efficiency: VLAs require 100K–1M+ demonstrations. Collecting robot data is expensive and slow. Active research: simulation-to-real transfer, foundation model pre-training, few-shot adaptation.