ChatGPT

# Vision-Language Navigation (VLN) vs Traditional Indoor Navigation

## Vision-Language Navigation (VLN) in Indoor Robotics

**Vision-and-Language Navigation (VLN)** is a paradigm where an embodied agent (a robot) navigates based on **natural language instructions** and what it **sees** in the environment [1]. The goal is to enable robots to follow human-like directions in indoor spaces, rather than requiring pre-programmed routes or coordinates. For example, a user might tell a home robot, *“Go to the living room and check if the front door is closed.”* The VLN-enabled robot must **understand the instruction’s meaning**, navigate through rooms and hallways, and visually verify the door’s state [2]. This requires combining **language understanding** (e.g. recognizing “front door”) with **vision-based navigation** (e.g. recognizing the door and the path to it). VLN is motivated by the vision of natural human-robot interaction – instead of using technical maps or coordinates, we can instruct robots just as we would a person [3].

### Core Principles of VLN

At its core, VLN is a **multimodal** problem: the agent receives **textual instructions** and **visual observations**, and it must produce a sequence of **navigation actions** to reach a goal described by the instructions [4]. Typically, the robot’s **inputs** are images from an onboard camera (e.g. an egocentric view or panoramic views of its surroundings) and a **natural language instruction** (one or more sentences describing where to go). The **output** is a series of actions or motion commands (e.g. “move forward”, “turn left”) that move the robot along the described route [4]. Unlike traditional methods, VLN generally does *not* assume a prior map of the environment – the agent is expected to **learn** how to interpret instructions and navigate by relying on its visual sensor data and prior training. Internally, VLN models use AI techniques (often deep learning) to **align language with visual cues**. For instance, a VLN model might use a neural network that encodes the instruction (perhaps with an RNN or transformer for language) and another that processes the camera image (e.g. a CNN for vision); these features are combined in a policy network that decides the next action. A common approach is to use an **attention mechanism** so that the agent can focus on parts of the instruction that relate to what it currently sees [5]. This helps handle instructions referencing landmarks – e.g. an instruction *“Turn right after the sofa”* requires the agent to visually identify the **sofa** and know when it has passed it, in order to execute the *“turn right”* at the correct moment [5]. In summary, VLN’s principle is to learn a mapping from (Vision, Language) → Actions, so the robot can **“ground”** language in the physical world (connecting words like “sofa” or “door” to the actual objects and locations it perceives) and navigate accordingly.

### Typical Inputs, Tasks, and Examples

**Inputs:** A VLN agent primarily relies on an **RGB camera** (sometimes along with depth sensors) for vision, and it receives a **natural language instruction** describing a navigation task. The language input could be a detailed route (*“Exit the bedroom, turn left down the hallway, and stop at the painting on the right”*) or a high-level goal description (*“Go to the kitchen and find the fridge”*). The agent may also maintain an **internal state**

**or memory** of past observations (to avoid getting lost), but it doesn't start with a metric map of the area – it learns to build a mental model of the environment as needed [1].

**Navigation tasks:** VLN models are designed for tasks where following spoken or written directions is required. A common task formulation is **instruction-following**: the agent is given a single, self-contained set of directions to go from a start point to a target location. For example, the Room-to-Room (R2R) benchmark provides instructions like "**Go out of the bedroom, down the hallway, then enter the second door on your left**," and the agent must end up at the correct destination. Another type is **goal-oriented instructions**, where the language describes the goal rather than the route (e.g. "**Find the bathroom and go inside**" – the agent must infer how to get there). VLN tasks can be further categorized by interaction style. In **single-turn** tasks, the instruction is given once at the beginning, and the agent must carry it out to completion [6]. In **multi-turn** or dialog-based VLN, the agent may have back-and-forth communication with a human, receiving incremental instructions or asking questions (for instance, a human might correct the course or answer the agent's questions if it gets confused). Some research tasks allow the agent to ask for help if an instruction is ambiguous, making it an interactive guiding scenario. These variations demand different capabilities: a route-oriented instruction requires the agent to sequentially follow step-by-step directions, while a goal-oriented task tests the agent's ability to navigate towards an objective using environmental cues (e.g. knowing that a "kitchen" might contain a fridge or certain furniture). **Figure 1** below illustrates an example of a VLN scenario where a robot is given a complex instruction and must parse it into a path:

[7] *In Vision-Language Navigation, an agent uses language and vision together to decide its path. In this example, the instruction (*"First move to the plant, then 3 meters south, then go between the keyboard and the bowl"*) contains references to objects and spatial directions. A VLN system must locate the mentioned landmarks ("the plant," "the keyboard," "the bowl") and understand spatial cues ("3 meters south," "between ...") to plan and execute the route. The path planned by the robot (shown as a sequence of colored lines/points) reflects how the robot interprets the instruction, turning natural language into navigable actions.* [7]

**Training and learning:** VLN models are usually trained on **annotated trajectories** in simulated or real environments. For example, a dataset might consist of human-written instructions paired with the paths a person would take in a house or virtual environment (Matterport3D scans are commonly used in VLN research [8]). The agent learns by imitation (following the examples) or by reinforcement learning (gradually improving by trial and error to reach the correct goal). Because VLN involves understanding free-form language and complex visuals, these models tend to be large neural networks that require a lot of data. Generalization is a key challenge: a good VLN agent should handle **previously unseen environments** and novel instructions it wasn't explicitly trained on [1]. Recent research has improved VLN performance by using **pre-trained vision-language models** (like CLIP or large language models) to give the agent prior knowledge of objects and language [9] [10]. Nonetheless, VLN in real physical robots is still an evolving area – most demonstrations happen in simulators or controlled indoor spaces.

## Traditional Indoor Navigation (AMCL + Navigation2)

Traditional indoor navigation approaches rely on **explicit mapping, localization, and path planning algorithms** – a model-based paradigm that has been the foundation of mobile robotics for decades. A representative pipeline is the ROS **Navigation Stack** (in ROS 2, this is often referred to as **Navigation2** or **Nav2**), which combines **Adaptive Monte Carlo Localization (AMCL)** for pose estimation with global and local planners for motion control. In this approach, before the robot moves autonomously, it usually needs a **map** of the environment (often a 2D floorplan map obtained via SLAM or provided by the user). The robot's task is framed as: "Given my current pose on the map and a desired goal pose on the map, compute and execute a safe path to the goal." All inputs and outputs are defined in terms of this map and geometric coordinates, rather than high-level concepts.

## Adaptive Monte Carlo Localization (AMCL) – Localization on a Map

**Adaptive Monte Carlo Localization** is a **probabilistic localization** algorithm that allows a robot to estimate its own position and orientation (together called the "pose") within a known map [11] [12]. AMCL uses a **particle filter**, which means it represents the robot's uncertain pose as a cloud of many hypothetical poses ("particles"). Each particle has a weight indicating how likely it is to be the true pose. The algorithm works as follows: initially, particles are spread across the map (if the robot's starting location is unknown, they might be spread everywhere; if you have a rough idea of start pose, they can be initialized around that). As the robot **moves**, it uses its **odometry** (wheel sensors/IMU) to shift all particles accordingly, and as the robot **senses** (typically using a 2D **LiDAR** scanner or depth camera), AMCL compares the sensor data to the map. Particles that align well with the observed landmarks or walls (i.e. a particle's pose would produce a similar laser scan on the map) are given higher weight, while those that don't match are down-weighted or discarded [13]. AMCL then resamples particles based on these weights, concentrating particles in the more likely regions. Over time, the particle cloud **converges** to the robot's true pose – essentially, the robot "figures out" where it is on the map by eliminating pose hypotheses that don't agree with sensor readings. The method is **adaptive** because it can adjust the number of particles on the fly and inject random particles if needed (for example, if the robot gets *"kidnapped"* or lost, AMCL will globally respread particles to recover) [14]. The output of AMCL is a continuously updated best estimate of the robot's pose on the map (usually a coordinate (x, y) plus orientation) [15]. This pose estimate is crucial: all subsequent planning happens in the map's coordinate frame, assuming the robot knows where it is. AMCL requires the environment map to remain largely the same as the one provided – significant changes (like moved walls or furniture) can degrade localization. However, for static indoor environments, AMCL is very robust and is a **standard solution** for localization in ROS and many commercial robots [16] [12].

## Navigation2 and the Classical Navigation Pipeline

Once localization is in place, the **Navigation2** framework (or the analogous ROS1 Navigation Stack) handles **path planning and control**. The navigation system loads the known **occupancy grid map** (a grid representation of the environment where free space and obstacles are marked) and uses it for planning. A user (or higher-level program) gives a **goal location** – typically this is a 2D coordinate (and optional heading) on the map, often obtained by clicking on a map or sending a goal message. The navigation stack will then:

1. **Global Planning:** Compute an optimal or feasible path from the current pose to the goal on the map. This is usually done with algorithms like A* or Dijkstra's algorithm on the grid (treating the map as a graph of free cells) or other planners for more complex robots. The output is a path (a sequence of waypoints through free space) [17]. A global planner needs the entire map but is typically static – it plans assuming the map's known obstacles.

2. **Local Planning and Obstacle Avoidance:** As the robot starts moving along the global path, a **local planner** (also called a controller) takes over. It looks at local sensor data (e.g. the latest laser scan) and creates a short-term trajectory to follow the global path while avoiding any new obstacles (like

people or chairs that weren't in the static map) [18] [19]. This involves building a **local costmap** around the robot – a short-range map that marks obstacles detected by sensors – and using methods like the Dynamic Window Approach (DWA) or model-predictive control to compute velocity commands that move the robot towards the next waypoint without collisions.

3. **Motion Execution:** The local planner outputs **velocity commands** (e.g. linear and angular speed) to the robot's motors at a regular control frequency [20]. The robot drives, constantly updating its pose via AMCL and updating sensor readings; the cycle of planning and control repeats until the goal is reached. Navigation2 uses a **feedback loop**: if something goes wrong (e.g., the path is blocked), recovery behaviors can kick in (like replanning a new path, rotating in place to find a path, or other scripted behaviors). All of these modules are coordinated by a behavior tree or state machine in Navigation2, making the system modular and resilient [21] [22].

*Architecture of a typical classical navigation stack (ROS Navigation). The system is highly modular. A map server provides the static occupancy grid map of the environment, and AMCL continuously localizes the robot on this map (using sensor data like laser scans). A global planner computes a path from the robot's current pose to the goal pose on the map (usually an optimal path on the free-space grid). A local planner (controller) then generates real-time motor commands to follow the path, using a local costmap built from recent sensor data to avoid obstacles. All components communicate via the ROS framework. This structured pipeline is an example of a model-based approach: it explicitly uses a map and physical sensor models to achieve reliable navigation [17].*

**Inputs and outputs:** The traditional pipeline requires specific inputs: **a map**, plus live **sensor streams** and **odometry**. Common sensors are 2D LiDARs for detecting distances to walls/obstacles (matched against the map for localization and used in the costmap for avoidance) and wheel odometry for tracking movement; some robots use visual SLAM or depth cameras, but the system fundamentally expects geometric data. The **goal** is given in map coordinates (for example, "goal: (x=4.2, y=-1.3) in the map frame"). There is no native concept of "kitchen" or "sofa" in this framework – the goal has to be a coordinate or a pre-defined named waypoint. The **output** of the navigation stack is the low-level velocity commands (`cmd_vel` in ROS) that drive the wheels toward the goal [20]. Throughout the process, the robot maintains a transform between coordinate frames (e.g. **map frame** to **base frame** via an **odom frame**), ensuring all components share a common reference for where things are. Navigation2 is designed to be **platform-agnostic** – it can work with different robot types (differential drive, car-like, legged, etc.) as long as the appropriate kinematic model is considered in the planners [23] [20]. It also provides many **tunable parameters** (for speeds, acceleration limits, obstacle margins, etc.) to customize behavior. The result of this classical approach is a **reliable, predictable navigation**: if the map is accurate and the environment doesn't change radically, the robot will follow the planned path and reach exactly the coordinate specified. This approach has been proven in many real-world deployments (from **warehouse AGVs** to **service robots in offices**), and it's considered **production-grade** – for instance, Nav2 is described as a high-quality framework used by 100+ companies [24].

## Key Differences: VLN vs. Classical Navigation

Now, how do VLN and traditional navigation compare? They approach the same ultimate problem – moving a robot to a desired location – in fundamentally different ways. Below are the **key differences** in approach, requirements, and use cases:

- **User Instructions vs. Goal Specification:** *VLN accepts human-like instructions, whereas classical navigation requires formal goals.* In VLN, the **input command is a natural language instruction** (e.g. "*Go past the elevators and find the conference room*"), which the robot parses and follows. By contrast, a traditional system needs a **precise goal** in the robot's coordinate system – usually an *(x, y)* position on the map or a series of waypoints. Classical navigation does not inherently understand language; any spoken or written command must be translated by an external process into a map location or a sequence of predefined actions. This means VLN is much more **flexible in how a user can instruct the robot**. A person can refer to landmarks or use vague terms in VLN, and if the model is trained for it, the robot can handle it (e.g. understanding "elevator" or "conference room" as concepts). Traditional navigation, on the other hand, excels if you have a specific target: e.g., "go to coordinate (10, 5)" or "go to Charging Station A." There have been systems that bridge this gap by mapping place names to coordinates (for example, a developer might label a spot on the map as "conference room" so that a voice command can trigger a goal there), but that's still fundamentally using the classical approach under the hood. **In summary:** VLN's strength is an intuitive interface (instructions in English or other languages), whereas classical navigation is driven by explicit, pre-defined goal locations [25].

- **Map Requirement and World Knowledge:** *Classical navigation is map-dependent and model-based, while VLN is map-free (on the fly) and model-free.* A traditional AMCL+Nav2 robot **must have a map** of its environment to navigate; the map is its world model, and all planning is done with respect to that map [25]. Building this map often requires a prior exploration (using SLAM) or obtaining floor plans. If the environment changes (furniture moves, a new obstacle appears), the map can become outdated, though the local planner can handle *temporary* obstacles to some extent. VLN methods, in contrast, do not start with a metric map. They attempt to navigate using **first-person observations and learned knowledge**. In essence, a VLN agent carries its "world model" in the weights of a neural network, not in a discrete occupancy grid. This makes VLN potentially more **adaptable to new environments** – it can encounter a completely new house and still follow instructions there, if it has learned generally what hallways, kitchens, doors, etc., look like. However, VLN agents can struggle with **spatial precision and unseen layouts**: without a perfect map, they might get disoriented more easily or fail to find an object that wasn't in their training data. Meanwhile, the classical robot will have no trouble as long as the new environment is mapped; but if you drop it into an unmapped space, it must **first map it** (or you must drive it around to create a map) before it can navigate autonomously. In summary, classical navigation is **model-based** (explicit map, known sensor models) and tends to be very robust *after* upfront mapping, whereas VLN is **model-free** in execution (no given map) but requires learning from a lot of data to internalize how to navigate [25]. This also means classical methods have strong performance guarantees in known maps, while VLN's success is probabilistic – it might fail in novel scenarios or underperform if instructions are phrased in an odd way.

- **System Architecture and Transparency:** *Traditional navigation is modular and explainable; VLN is end-to-end learned and opaque.* The classical pipeline is **highly modular** (localization, mapping, planning, control are separate) [17]. Each module can be tuned or replaced independently, and engineers can inspect things like the map, the planned path, or the particle cloud in localization to understand what the robot is doing. This modular design also means errors can often be pinpointed (e.g., "the localization failed" vs "the planner got stuck"). VLN systems, on the other hand, often use a single integrated model (or a tightly-coupled set of learned models) that directly maps sensory input and language to actions. This is often described as a "**black box**" – it's not trivial to interpret *why* the agent made a certain decision, since it's encoded in neural network weights rather than clear rules. For example, if a VLN agent takes a wrong turn, it could be due to a subtle misunderstanding of a word or mis-recognition of an object, which is hard to debug. The benefit of the learned, end-to-end architecture is that the agent can, in theory, discover **optimal behaviors or features** that humans might not program – it can learn to pay attention to visual details or language phrasing nuances automatically. It also tends to integrate perception and decision-making: a VLN agent might learn a representation that simultaneously captures "where am I and what did the instruction ask," rather than maintaining a separate map and doing pathfinding on it. This can be advantageous in complex, unstructured scenarios. However, **engineering-wise, classical navigation is easier to validate** (you can mathematically guarantee the planner finds a path if one exists, etc.), whereas VLN requires extensive testing to gain confidence. Some newer systems try to get the best of both worlds by making VLN more modular (for example, using a learned **language understanding module** to extract waypoints, then using a classical planner to execute them). Conversely, classical stacks are also evolving by adding semantic layers (like recognizing objects on the map) to better handle higher-level requests [26] [27].

- **Sensors and Perception:** *Classical methods rely on precise range and odometry sensors; VLN leans heavily on visual perception.* In a typical AMCL+Nav2 setup, the primary sensor modality is a **range sensor** (2D LiDAR is common) for mapping and localization. Vision (cameras) might not be used at all in the core navigation (though a robot could have cameras for other purposes). The LiDAR gives precise distance measurements to obstacles, which is ideal for building geometric maps and avoiding collisions. **VLN, by definition, requires vision** – the "Vision" in VLN usually means an RGB camera feed that the agent uses to recognize landmarks and understand its environment visually. VLN agents are often tested in photorealistic simulated houses where the only observations are images. They must interpret those images: e.g. recognize a **"sofa"**, **"door"**, **"plant"**, or other objects if mentioned in the instructions. This means VLN involves solving challenging computer vision problems (object detection, scene recognition) on the fly, whereas classical navigation simplifies the world into abstract obstacles and free space. The advantage for VLN is **semantic understanding**: it can make decisions based on object types and semantic cues (e.g. "turn at the **red chair**" or "go to the **kitchen**" – a VLN agent can be trained to know what a kitchen looks like). Traditional navigation has no built-in concept of a "kitchen"; it only knows about distances and obstacles, unless a human labels the map with that information. On the flip side, classical approach benefits from **metric accuracy**: sensors like LiDAR and wheel odometry provide exact positions and distances, so the robot can navigate with centimeter-level precision and will, say, stop exactly 0.5 m in front of an obstacle because the algorithms are designed for that. VLN's reliance on vision can make it less precise (images might be ambiguous for depth or exact scale) and more susceptible to lighting or sensor noise. Many VLN setups do use an **RGB-D camera** (color + depth) to get some depth information, and some even incorporate compass or GPS-like data for orientation, but the heavy lifting is still done by vision and language understanding. In practice, an indoor service robot might combine the two: use classical navigation for moving safely and use a language/vision module to decide where to go or to recognize targets. But in a pure comparison, **classical = geometry-driven, VLN = semantics-driven**.

- **Learning vs. Planning (Model-Free vs. Model-Based):** Another perspective on their differences is through the lens of AI paradigms: **traditional navigation is a "planning" problem**, while **VLN is a "learning" problem**. The AMCL+Nav2 stack doesn't learn from experience (aside from minor self-tuning in adaptive algorithms); it will do the same thing every time given the same inputs, following pre-programmed algorithms. It's a **model-based approach** – the robot's behavior is derived from an explicit model of the environment (the map) and formal algorithms (e.g. shortest path planning, control theory for driving). VLN, in contrast, uses **machine learning models trained on data**. The robot's behavior is optimized during a training phase, and what it "learned" is stored in the model parameters. This makes VLN analogous to a **model-free approach** in reinforcement learning: the robot doesn't calculate an optimal path on a known map; instead, it tries actions and learns what tends to lead to success. One consequence is that **VLN requires a lot of data** and training time. Researchers must collect many example navigation instructions and trajectories (or even synthesize them) to teach the model [28]. If the VLN model is large (sometimes using transformers or LLMs), it may even be pretrained on broad vision-language data before fine-tuning on navigation tasks [9]. Classical navigation requires no such training – the "intelligence" is in the algorithms. The trade-off comes in adaptability: a trained VLN agent might handle scenarios that a human didn't explicitly program for. For instance, a VLN agent might learn to implicitly **navigate around a chair** that's mentioned as a landmark ("go past the chair…" implies it should not collide with it) even if it wasn't hard-coded as an obstacle avoidance rule. A classical robot, however, would rely entirely on its obstacle avoidance for that and might not understand the instruction at all. On the other hand, if something outside of its training distribution occurs, a VLN agent can fail strangely – e.g., an instruction phrased in a way it hasn't seen, or a totally new object reference, might confuse it. The classical system will **never** misunderstand an instruction it can't parse – it simply doesn't handle it – but if given a clear goal, it will consistently execute it. In short, **classical navigation's reliability vs. VLN's flexibility** is a classic example of planning vs learning. There is ongoing research on combining these, such as using learning to improve parts of the navigation stack or using symbolic planners with language; but pure VLN methods are generally considered at an early stage for real-world deployment, whereas classical methods are a mature technology [28].

- **Use Case Suitability:** Given their differences, VLN and traditional navigation are suited to different scenarios (at least with current technology). **Traditional navigation (AMCL + Nav2)** is excellent for applications where the environment can be mapped and the tasks are well-defined. For example, in a warehouse or hospital, robots often repeat routes transporting goods – here a mapped approach with predefined goal points is highly efficient and dependable. Setup effort (mapping, tuning) is worthwhile because the environment is relatively stable and the robot will use the same map for a long time. These systems also integrate well with safety sensors (like laser scanners) to avoid collisions, and they have predictable behavior (important for safety certification and trust). **VLN** comes into play for more *open-ended, interactive scenarios*. Imagine a home assistant robot: every user's house is different, and people will ask the robot to do varied things. It's impractical to pre-program all possible goals on a map or require the user to tag every object/location. VLN promises that you could say, *"Robot, please go to the kitchen and bring me a snack from the counter,"* and the robot would handle it – navigating to what it recognizes as the kitchen, finding the "counter" and identifying a snack. This level of semantic understanding is beyond current traditional navigation. Similarly, VLN could be useful in **office tour guide robots** (following spoken directions from a visitor), or **search-and-rescue** scenarios where a human could guide a robot with natural language in a partially known environment. However, VLN in practice is still mostly a research domain. Its current deployments are limited, whereas classical navigation is **field-proven**. We are beginning to see hybrids: for instance, recent work on **Visual Language Maps (VLMaps)** combines the idea of a map with learned semantic features, allowing a robot to build a map that can be queried with language [26]. Such a robot can leverage the reliability of mapping (it knows space and distances) while also understanding open-vocabulary commands (the map is annotated with features like "this area has a chair") [26] [27]. This kind of approach hints that the future of indoor navigation may not be an *either-or* choice: robots might use classical algorithms for low-level control and safety, while using VLN-style language understanding at a higher level to decide where to go or to interpret human instructions.

In conclusion, **Vision-Language Navigation and traditional navigation differ fundamentally in approach**. VLN is **data-driven**, seeking to imitate human instruction-following using AI; it excels at interpreting flexible, semantic requests but currently lacks the guaranteed reliability of classical methods. Traditional navigation is **model-driven**, treating navigation as a geometric problem; it provides robust and precise movement in known spaces but requires structured input (maps and coordinates) and cannot on its own understand human language or intents. For someone learning about indoor robotic navigation, VLN represents the frontier of making robots more **intuitive and human-compatible**, whereas AMCL+Nav2 represents the **tried-and-true foundation** that underpins most real indoor robots today. Both are important: the classical approach ensures a robot can move from point A to B safely, and the VLN approach aims to make it so you can tell the robot **which** point B you mean using natural, friendly terms. Together, they move us closer to indoor robots that are both competent navigators **and** easy to command.

**Definitions Quick Reference:**
- **Embodied Agent:** A robot or virtual agent that has a physical presence (or simulated body) in an environment and can perceive and act within that environment [3]. VLN is studied on embodied agents that see and move, not just disembodied AI.
- **Occupancy Grid Map:** A grid-based representation of space where each cell is marked free or occupied (or probability of occupancy). Used in classical navigation for path planning and localization.
- **AMCL (Adaptive Monte Carlo Localization):** A algorithm that uses a particle filter to localize a robot on a known map. "Adaptive" refers to adjusting the number of particles and respreading them as needed (e.g., to recover from localization errors) [12].
- **Navigation2 (Nav2):** The ROS 2 Navigation framework, containing components for mapping, localization, planning, and control. It's the successor to ROS 1's navigation stack and enables mobile robots to autonomously navigate to a given goal pose [23] [17].
- **Model-Based vs. Model-Free:** In robotics/AI, *model-based* methods rely on a known model of the world or the agent's dynamics (here, a known map and kinematic models), while *model-free* methods learn a policy without an explicit environment model (VLN agent learns how to navigate without a provided map, effectively learning its own representation).
- **Particle Filter:** An algorithm for estimating a probability distribution over some state (like robot pose) using a set of random samples (particles). Each particle is a hypothesis of the state, and the set is updated (weighted and resampled) based on sensor data to gradually approximate the true state [13].
- **SLAM:** Simultaneous Localization and Mapping, a process where a robot builds a map of an unknown environment while tracking its pose. SLAM can be seen as the precursor step to using AMCL (which assumes the map is known).
- **Waypoint:** An intermediate location or milestone on a route. In classical navigation, a global path is often

a sequence of waypoints. In VLN, sometimes instructions contain waypoints implicitly (landmarks or specific turns).
- **Local Planner / Global Planner:** In navigation stacks, the global planner plans an overall route on the global map, and the local planner handles immediate motion and obstacle avoidance using local sensor data [18].
- **Vision-Language Model:** A machine learning model (like CLIP) that can connect visual data with language, often used in VLN to help recognize described objects or scenes [9].

---

[1] [4] [6] arxiv.org
https://arxiv.org/pdf/2108.11544

[2] [5] Vision And Language Navigation. This article briefly summaries the... | by AI Club @IIITB | AI Club @IIITB | Medium
https://medium.com/ai-club-iiitb/vision-and-language-navigation-3c90ecc1044c

[3] [8] [9] [10] Vision-and-Language Navigation Today and Tomorrow: A Survey in the Era of Foundation Models
https://arxiv.org/html/2407.07035v1

[7] [25] [26] [27] [28] Robot navigation with vision language maps - viso.ai
https://viso.ai/deep-learning/robot-navigation/

[11] [12] [16] amcl - ROS Wiki
http://wiki.ros.org/amcl

[13] [14] [15] 7. AMCL Adaptive Monte Carlo Localization
http://www.yahboom.net/public/upload/upload-html/1722937503/7.%20AMCL%20Adaptive%20Monte%20Carlo%20Localization.html

[17] [18] [19] [20] [21] [22] [23] [24] Nav2 — Nav2 1.0.0 documentation
https://docs.nav2.org/