Building a depth-based grasp planning system for assistive robotics

Start with a Python-native architecture using Open3D for perception and either AnyGrasp or GraspNet-1Billion for grasp planning. This approach achieves a working prototype in your 1-month timeline, leverages your team's Python/AI expertise without requiring ROS experience, and provides a clear migration path to ROS2 when you need advanced features. ROS2 brings significant value for production systems but would consume your entire timeline just learning the framework.

This recommendation is based on analysis of your specific constraints: a 2-person team with strong AI/coding skills and mechanical engineering expertise but no ROS experience, testing on Windows and macOS platforms, and aggressive timeline targeting a working prototype in approximately 1 month. The assistive robotics use case for ALS patients demands reliability over complexity, making a focused Python implementation more appropriate than a full robotics middleware stack initially.

Current state-of-the-art in grasp planning for depth cameras

The field has converged around learning-based approaches that operate directly on point clouds from depth cameras, with **AnyGrasp and GraspNet-1Billion representing the current state-of-the-art** for your use case. These methods achieve 90-93% success rates on novel household objects—approaching human-level performance—and work explicitly with the RealSense D435 depth camera you're using.

AnyGrasp, published in IEEE Transactions on Robotics in 2023, achieves **93.3% success on 300+ unseen objects** with real-time performance and explicit RealSense D435 compatibility. The system generates 7-DOF dense grasp poses with temporal smoothness for tracking moving objects, trained on real perception data to handle depth sensor noise robustly. The commercial SDK requires license registration (typically 2-3 days approval) but provides the fastest path to production-quality grasping with minimal implementation effort.

GraspNet-1Billion serves as an excellent fully open-source alternative from the same Shanghai Jiao Tong University research group. Built on a dataset of 1 billion grasp poses across 190 scenes and 88 household objects, it provides RealSense-specific pre-trained models that work out of the box. The architecture uses

PointNet++ encoders with decoupled approach direction and operation parameters, achieving 85-90% success rates with clear documentation and active community support.

Contact-GraspNet from NVIDIA represents another strong contender, using a novel 4-DOF contact-based representation that reduces from 6-DOF for computational efficiency. Training in just 40 hours (versus a week for competitors) on 17 million simulated grasps from the ACRONYM dataset, it achieves efficient end-to-end inference in a single forward pass. PyTorch implementations and ROS wrappers are readily available, making integration straightforward.

Traditional geometric approaches like GPD (Grasp Pose Detection) and Dex-Net 2.0 remain viable for specific scenarios. GPD achieves 93% success in dense clutter without requiring training, using a pre-trained CNN to classify sampled grasp candidates from point clouds. Dex-Net 2.0 from Berkeley operates on depth images rather than full point clouds, achieving 93% success on known objects with 0.8-second planning time and the unique advantage of CPU-viable operation. However, both require more manual tuning and lack the seamless segmentation integration of modern learning-based methods.

Recent 2024-2025 developments emphasize language-driven grasping, enhanced scale handling for small objects, and NeRF-based methods for multi-view reconstruction. Most relevant for your project are the **AnyGrasp SDK updates in 2024** adding dense prediction mode and Python 3.10 support, along with Scale-Balanced GraspNet improvements for detecting small objects like utensils and medication bottles—critical for assistive applications.

For your 1-month timeline with a team lacking ROS experience but possessing strong AI/coding skills and mechanical engineering expertise, **AnyGrasp emerges as the primary recommendation** with GraspNet-1Billion as backup. Both offer pre-trained models eliminating training time, excellent RealSense compatibility, and straightforward integration with RF-DETR segmentation. Implementation difficulty ranks as LOW-MEDIUM, requiring primarily API integration rather than algorithm development. Avoid custom training any method, GIGA's complex architecture, or GPD's extensive dependencies within your initial timeline.

Architecture comparison: choosing between Open3D, ROS2, and Python toolkits

---

The architectural decision fundamentally determines whether you'll achieve a working prototype in 1 month or spend that time learning frameworks. **For your specific constraints, a Python-native architecture using Open3D wins decisively**, with ROS2 deferred to a later migration when you need advanced motion planning, simulation, or multi-sensor integration.

Why Python-native with Open3D succeeds for rapid prototyping

Open3D provides native RealSense SDK v2 integration (v0.12+) without separate librealsense installation, works seamlessly on Linux, macOS, and Windows without platform-specific issues, and integrates directly with your existing Python stack. The learning curve measures just **3-7 days to productivity** for a team with Python/AI experience, compared to 3-4 weeks for ROS2. Your entire existing stack—Flet GUI, pyrealsense2, xarm-python-sdk, and RF-DETR—connects through direct Python imports without middleware or message passing overhead.

The data flow remains straightforward: RF-DETR generates object masks, RealSense provides depth frames, Open3D creates point clouds with mask-based cropping, grasp detection runs on object-specific point clouds, and xarm-python-sdk executes the selected grasp. All components run in a single Python process with shared memory, enabling efficient GPU utilization for RF-DETR inference and point cloud processing simultaneously.

While Open3D lacks built-in grasp planning libraries, it provides an excellent foundation for integrating standalone grasp detectors. You can integrate GPD as a C++ library with Python bindings, implement custom antipodal grasp detection following MIT's manipulation course examples, or use AnyGrasp/ GraspNet-1Billion which already output standard grasp representations. The MIT "Manipulation: Perception, Planning and Control" course at manipulation.csail.mit.edu provides production-ready geometric grasp planning code that integrates seamlessly with Open3D.

For motion planning and kinematics, the Robotics Toolbox for Python provides forward/inverse kinematics, Jacobians, and trajectory generation with 30+ robot models included. Alternatively, implement simple IK using the xArm SDK's built-in Cartesian positioning for initial prototyping. Most pick-and-place tasks for assistive robotics don't require sophisticated motion planners initially—linear interpolation with obstacle checking suffices for month-one demonstrations.

Why ROS2 fails the 1-month timeline

ROS2 represents the industry standard for production robotics systems, offering MoveIt2 for advanced motion planning, comprehensive sensor ecosystem integration, excellent simulation support with Gazebo, and the largest community in robotics. However, for a team without ROS experience targeting a 1-month prototype, **ROS2 would consume your entire timeline learning concepts rather than building your application**.

The fundamental barrier is conceptual overhead: publishers/subscribers for inter-process communication, services/actions for synchronous operations, transforms (TF2) for coordinate frame management, launch

files for system initialization, and the colcon build system. Debugging distributed systems proves significantly harder than single-process Python applications. Testing requires understanding ROS bags, RViz visualization tools, and the overall message-passing paradigm.

Platform constraints present another critical issue. **macOS support effectively died after ROS2 Foxy in May 2021**, forcing Mac users onto Linux VMs or dual-boot configurations. Windows maintains Tier 2 support—functional but with limited package availability and no official binary releases for many packages. If team members develop on Macs, this alone makes ROS2 impractical for your timeline.

The learning curve estimates at **3-4 weeks to basic productivity** for those new to ROS. You'd spend week one understanding ROS2 fundamentals, week two fighting installation and dependencies, week three building basic perception pipelines, and week four attempting integration—arriving at the deadline without a working system. For comparison, the Python-native approach achieves a working end-to-end demo by week two.

When ROS2 makes sense: You should plan migration to ROS2 in months 3-6 when you encounter needs for sophisticated collision-aware motion planning beyond basic IK, integration of multiple sensors or robots, comprehensive simulation for testing without hardware, or when scaling beyond single-arm pick-and-place. The Python-native architecture provides clear module boundaries that enable gradual migration—wrap your perception pipeline as ROS2 nodes, add MoveIt2 for planning, switch to xarm_ros2 for control, while keeping working code with minimal disruption.

Python robotics toolkit options

The Robotics Toolbox for Python (Peter Corke, Queensland University of Technology) provides forward/inverse kinematics, Jacobians, trajectory generation, robot models for 30+ platforms, and Swift visualization using Three.js—all in pure Python. The learning curve spans just **1-2 weeks to basic proficiency**. While it lacks built-in grasp planning, it excels at the motion planning and kinematics layer between grasp detection and arm control.

PythonRobotics (Atsushi Sakai) serves primarily as an educational resource and algorithm reference rather than a production framework. It covers path planning (RRT, *PRM*, *A*), trajectory tracking, and localization with clear implementations, but isn't designed for real-world system integration.

For grasp simulation and validation, **PyBullet** provides physics simulation with built-in depth rendering, fast real-time performance, Python-native API, and setup time under 1 hour. This enables testing grasp strategies before hardware deployment without the complexity of Gazebo or Isaac Sim. Several research projects (VGN, GraspNet) include PyBullet integration examples you can adapt.

Recommended hybrid strategy

Phase 1 (Month 1): Python-native prototype

```
Open3D (perception + point cloud processing)
+ AnyGrasp SDK or GraspNet-1Billion (grasp planning)
+ Robotics Toolbox or xArm IK (kinematics)
+ xarm-python-sdk (control)
+ Flet (GUI)
+ PyBullet (simulation/testing)
```

This achieves your working prototype demonstrating camera capture → segmentation → grasp detection → arm execution with appropriate safety systems.

Phase 2 (Months 2-6): Selective ROS2 integration

Begin migrating components when benefits justify effort. Wrap your Python perception module as a ROS2 node publishing point clouds, add MoveIt2 when you need sophisticated motion planning, switch to xarm_ros2 for standardized control interfaces, but keep working perception code largely unchanged. This gradual migration spreads learning curve over months while maintaining a functional system throughout.

The migration cost estimates at **2-4 weeks of refactoring** when you're ready, but can proceed incrementally—one module at a time rather than big-bang rewrite.

Detailed architecture comparison matrix

Criterion	Open3D Python-Native	ROS2	Robotics Toolkit
Time to prototype	✅ 2-3 weeks (90% confidence)	❌ 6-8 weeks minimum (30% confidence)	✅ 3-4 weeks (85% confidence)
Learning curve	✅ 3-7 days	❌ 3-4 weeks	✅ 1-2 weeks
Platform support	✅ Win/Mac/Linux	❌ Linux only (Mac dropped)	✅ Win/Mac/Linux

Criterion	Open3D Python-Native	ROS2	Robotics Toolkit
Existing stack integration	✅ Direct Python imports	⚠️ Requires wrappers	✅ Direct Python
pyrealsense2	✅ Native support	⚠️ Use realsense-ros	✅ Direct
xarm-python-sdk	✅ Direct calls	⚠️ Use xarm_ros2	✅ Direct
Flet GUI	✅ Same process	❌ Separate process	✅ Same process
RF-DETR	✅ Native PyTorch	⚠️ Needs wrapper	✅ Native
Grasp planning	⚠️ Integrate external	✅ MoveIt grasps	⚠️ Implement custom
Motion planning	⚠️ Basic IK	✅ MoveIt2 excellent	✅ Robotics Toolbox
Simulation	⚠️ PyBullet separate	✅ Gazebo integrated	⚠️ PyBullet
Community size	Medium	✅ Very large	Small-Medium
Documentation	✅ Excellent	✅ Extensive	✅ Good
Future extensibility	⚠️ Manual scaling	✅ Ecosystem support	⚠️ Limited
GPU utilization	✅ Simple management	⚠️ Multi-process	✅ Simple
1-month verdict	✅✅✅ Recommended	❌ Too slow	✅✅ Viable

Step-by-step technical implementation approach

Week 1: Establish the perception pipeline

Install Open3D (`pip install open3d`), pyrealsense2, and verify RealSense D435 connectivity. Configure optimal depth settings: 848×480 resolution at 30 FPS with Medium Density preset for best

accuracy/speed balance in the 0.3-3.0m working range. Implement the critical filter pipeline—decimation, spatial (alpha=0.5, delta=20), temporal (alpha=0.4), and hole filling—to handle D435 depth noise robustly.

Integrate your existing RF-DETR segmentation to generate object masks. For each detected object, apply the mask to zero out background pixels in the depth image, then convert to a 3D point cloud using RealSense intrinsics. Use Open3D to visualize the segmented object point clouds and verify depth accuracy, which should measure approximately 10mm at typical working distances of 0.3-0.8m.

Implement point cloud preprocessing using Open3D: statistical outlier removal (20 neighbors, std_ratio=2.0), voxel downsampling if needed for performance, and normal estimation for grasp planning. Test this complete perception pipeline with 5-10 household objects relevant to your ALS use case—cups, bottles, utensils, phone, medication containers.

Deliverable: Visualize clean, segmented point clouds for target objects with RF-DETR masks correctly applied. This foundational perception pipeline will support all subsequent grasp planning work.

Week 2: Implement grasp detection

Apply for the AnyGrasp SDK license immediately (2-3 days approval time) while setting up GraspNet-1Billion as your backup. Clone the repository, compile PointNet++ operators with CUDA support, and download the RealSense-specific pre-trained model. Both systems integrate similarly: feed the masked object point cloud, receive ranked grasp poses with 6-DOF transformations (position, orientation) and quality scores.

For AnyGrasp, configure the SDK with your workspace boundaries (as tight as possible around your manipulation area), enable the collision detection flag, and use the object_mask mode for integration with RF-DETR segmentation. The dense_grasp flag generates more candidates when you need options for user selection.

For GraspNet-1Billion, use the baseline inference script with the RealSense camera model. The output provides grasp rectangles with approach vectors and gripper widths, which you'll need to convert to your robot's coordinate frame. Implement grasp quality scoring using the combined metric: antipodal score (1.5× weight), force closure check (1.0× weight), distance to center of mass (0.5× weight), and binary collision check.

Visualize grasp candidates using Open3D's geometry rendering—display the object point cloud with gripper meshes positioned at proposed grasp poses. This visualization proves critical for debugging

coordinate frame issues and verifying grasp quality before hardware testing. Test grasp detection on your household object dataset and aim for **at least 5 ranked grasps per object** with quality scores above 0.7.

Deliverable: Generate and visualize ranked grasp poses for each target object type, with clear quality metrics indicating which grasps are most likely to succeed.

Week 3: Bridge to robot control with calibration

Hand-eye calibration represents the **most critical step** determining overall system accuracy. Use the ArUco marker method: print a 6×6 ArUco board, mount it rigidly on your table, move the robot to 15-20 distinct poses while capturing images and recording robot poses. Use OpenCV's `calibrateHandEye` function with the Tsai method to solve for the camera-to-robot-base transformation matrix.

Expect calibration accuracy of 2-5mm translation and 1-2° rotation. Validate by placing an ArUco marker at a known position, detecting it with the camera, transforming to robot coordinates, and commanding the robot to point at that position. The end effector should arrive within 5mm of the marker.

Implement the coordinate transformation chain: pixel (u,v) plus depth → 3D camera coordinates using RealSense intrinsics → robot base frame using calibration matrix → end effector commands in the xArm SDK (which uses millimeters, not meters). Document every coordinate frame convention explicitly to avoid the most common robotics mistake.

Begin motion planning implementation. For the 1-month prototype, simple inverse kinematics using xArm's built-in Cartesian positioning suffices—the SDK provides `set_position(x, y, z, roll, pitch, yaw)` that handles IK internally. For grasp execution, implement a state machine: approach (100mm above target at slow speed 100mm/s) → descend (to grasp position) → close gripper → lift (200mm up) → transport → release. Add safety checks before each motion: verify positions within workspace boundaries, check for E-stop, validate speed limits.

Test with soft objects first—foam shapes, cardboard boxes, soft plastic bottles—at 10-20% of normal speed. Only after successful soft object testing should you attempt rigid objects.

Deliverable: Execute complete pick-and-place sequences on soft test objects with calibrated, safe motion control.

Week 4: Safety systems and integration polish

Implement comprehensive safety systems starting with the hardware emergency stop—a large red mushroom button hardwired to the xArm controller within easy reach. Add software backup monitoring

space bar or designated key press to trigger emergency stop, plus a watchdog timer that stops motion if the main loop hangs for more than 500ms.

Configure force and speed limits following ISO 13482 recommendations for personal care robots: maximum contact force 50N (transient) or 25N (sustained), maximum speed near humans 250mm/s, workspace speed up to 500mm/s, and approach speed 100mm/s. Set the xArm's acceleration (500mm/s²) and jerk limits (2000mm/s³) to ensure smooth, predictable motion.

Define safe workspace boundaries in three dimensions plus a patient exclusion zone—a cylindrical region around the patient's body where the robot must not enter. Implement checks before every motion command, rejecting any trajectory that would violate boundaries. Test the emergency stop thoroughly: press it during motion, verify immediate cessation, ensure safe recovery and return-to-home procedures.

Integrate your Flet GUI with the perception and control pipeline. Display the camera view with RF-DETR segmentation overlaid, show detected objects with confidence scores, visualize the selected grasp pose on the object, provide manual object selection and grasp confirmation buttons, display real-time status (idle/perceiving/planning/executing/error), and show clear emergency stop button. The interface should follow assistive robotics principles: **quality over speed, clear feedback, easy abort options**.

Test the complete end-to-end system on your target object set: cups, bottles, utensils, phone, medication containers. Measure and document success rates, failure modes, and execution times. Aim for **70%+ success rate on test objects** with safe, predictable motion as your month-one completion criteria.

Deliverable: Fully integrated system demonstrating camera capture → RF-DETR segmentation → grasp detection → safe robot execution with GUI control and comprehensive safety systems.

Realistic timeline assessment: what's achievable versus what to defer

---

Achievable in Month 1 (HIGH confidence)

Core perception and planning pipeline: RealSense D435 point cloud acquisition with proper filtering, RF-DETR integration for object segmentation, grasp pose generation using pre-trained models (AnyGrasp or GraspNet), coordinate transformation from camera to robot base, and visualization of all intermediate steps for debugging.

Robot control and execution: Hand-eye calibration achieving 2-5mm accuracy, basic inverse kinematics using xArm SDK, simple pick-and-place state machine (approach → grasp → lift → transport →

release), testing on regular household objects with matte surfaces, and operation at reduced speeds (50-100mm/s) for safety.

Safety and interface: Hardware emergency stop with software backup, workspace boundary enforcement, force and speed limits, basic Flet GUI for object selection and status display, and initial testing with soft objects before rigid items.

Success metrics: 70%+ grasp success on 10 test objects, safe operation with E-stop and limits verified, full pipeline demonstrated end-to-end, and clear documentation of the system architecture for future improvements.

Defer to Months 2-3 (MEDIUM difficulty)

Advanced grasp planning: Fine-tuning grasp quality metrics for your specific gripper, implementing multiple grasp strategies (top-down, side-grasp, pinch), advanced collision detection with full arm geometry, and grasp failure recovery with automatic retry of alternative poses.

Motion planning improvements: Sophisticated trajectory planning with obstacle avoidance, dynamic replanning during execution, optimization for smoothness and speed, and integration with Robotics Toolbox or MoveIt2 for complex scenarios.

Challenging objects: Transparent containers (fundamentally difficult for depth cameras), highly reflective surfaces (metallic objects), deformable objects requiring force sensing, and small objects (under 2cm) requiring precise grasping.

User experience enhancements: Multi-object handling sequences, learn-by-demonstration for custom tasks, voice command integration, and personalized grasp preferences based on user feedback.

Defer to Months 4-6+ (HIGH difficulty)

Machine learning improvements: Custom training on your specific objects and gripper, sim-to-real transfer learning, reinforcement learning for grasp optimization, and online learning from successes and failures.

Advanced assistive features: Dynamic object grasping (moving targets), bimanual manipulation tasks, compliant control for feeding assistance, and force-based insertion tasks (plugging in devices).

Production robustness: Full collision avoidance using depth data, predictive maintenance monitoring, formal safety certification for patient use, and FDA clearance or CE marking for medical device status.

Research-grade features: Learning-based approaches trained from scratch (requires months of data collection and training time), GIGA or other implicit representation methods (high implementation complexity), dexterous manipulation with multi-finger hands, and tactile sensing integration.

The **critical mistake to avoid** would be attempting any of the deferred items in month one. Teams consistently underestimate robotics integration time. The recommended approach prioritizes a working, safe baseline that demonstrates the complete pipeline, then systematically adds capabilities based on real-world testing and user feedback.

Grasp quality metrics and gripper-specific considerations

Modern grasp planning evaluates candidates using multiple complementary metrics. **Force closure** remains the gold standard—a grasp achieves force closure when it can resist arbitrary external wrenches through contact forces alone. Check this computationally by verifying that contact forces span the entire wrench space, typically using convex hull algorithms on the Grasp Wrench Space (GWS). Berkeley's Dex-Net introduces **epsilon quality** measuring robustness to uncertainty in object pose, friction coefficients, and contact positions—this metric directly correlates with real-world success rates.

The **antipodal score** verifies that surface normals at contact points face opposite directions with the gripper axis bisecting them. For parallel-jaw grippers, check that the angle between surface normals and the gripper closing direction exceeds 80°. Add **distance to center of mass** as a heuristic—grasps closer to the COM generally provide better stability and resist toppling during lifting.

Implement a combined quality function weighting these metrics:

```
quality = 1.5 × antipodal_score + 1.0 × force_closure + 0.5 × com_score
quality *= (1.0 if collision_free else 0.0)
```

The collision check acts as a binary gate—any grasp causing gripper-object or gripper-scene collision receives zero quality regardless of other metrics.

For **parallel-jaw grippers** like those on the UFactory Lite 6, prioritize antipodal grasps with friction cone validation. The friction cone angle depends on material properties—use μ=0.5 (friction coefficient) for typical household objects. Verify that contact normals lie within the friction cone to ensure force closure. Check gripper width against object dimensions: most parallel-jaw grippers provide 50-150mm range, which handles cups, bottles, and utensils well but may struggle with very large (over 120mm diameter) or very small (under 15mm) objects.

Vacuum grippers excel at flat surfaces but require different quality metrics. Evaluate surface planarity using principal component analysis on local point cloud patches—fit a plane and measure deviation of points from that plane, with deviations under 2mm indicating good suction candidates. Calculate available surface area within the suction cup radius, requiring minimum 300mm² for reliable sealing. Check surface angle—vacuum cups work best on horizontal or gently sloped surfaces (under 30° from vertical approach). Avoid porous materials (fabric, paper) and wet surfaces that prevent sealing.

Multi-finger dexterous hands (likely future upgrade) require solving for stable 3+ contact force closure. The Robotics Toolbox and GraspIt! simulator provide force closure algorithms for arbitrary finger configurations. Modern learning approaches like Dex-Net 4.0 and AnyDexGrasp handle multi-finger planning but require significantly more complex hardware and control systems—defer these to future iterations after mastering parallel-jaw grasping.

Integration pipeline: from segmentation to execution

The complete data flow follows this sequence: RF-DETR produces bounding boxes and instance masks for objects in the RGB image, the RealSense D435 provides aligned color and depth frames (ensure alignment using `rs.align(rs.stream.color)`), you apply segmentation masks to the depth image to extract object-specific regions, then convert masked depth to 3D point clouds using camera intrinsics.

For each segmented object, extract the point cloud: `object_mask = rf_detr_output['masks'][object_id]`, `depth_masked = depth_image.copy()`, `depth_masked[~object_mask] = 0`, then use the RealSense `pointcloud()` API or Open3D's `create_from_rgbd_image()` to generate 3D points. Clean the point cloud with statistical outlier removal to eliminate noise: `cl, ind = pcd.remove_statistical_outlier(nb_neighbors=20, std_ratio=2.0)`.

Feed the cleaned point cloud to your grasp planner (AnyGrasp or GraspNet), which returns a list of candidate grasps. Each grasp consists of a 4×4 transformation matrix encoding position and orientation, gripper width (for parallel-jaw), and quality score. Rank grasps by quality and filter those outside object mask regions or causing collisions with the scene.

Transform the selected grasp from camera coordinates to robot base coordinates: `grasp_base = T_base_camera @ grasp_camera`. This requires your calibrated transformation matrix from hand-eye calibration. Verify the transformed pose lies within safe workspace boundaries before proceeding.

Generate the motion plan using a simple state machine: compute approach pose (offset 100mm along grasp approach vector), command robot to approach pose at slow speed (100mm/s), descend along approach vector to grasp position, close gripper to computed width, verify grasp success (force sensor or

binary gripper state), lift 200mm vertically, transport to target location, and release gripper. Monitor the E-stop flag and workspace boundaries before each motion segment.

The **coordinate frame conventions** require explicit documentation: RealSense optical frame has Z forward (into scene), Y down, X right; robot base frame typically has Z up, X forward; end-effector frame depends on mounting. Use visualization in RViz or Open3D to verify transforms—render coordinate axes at each frame and manually check that transformations produce expected results.

Common integration issues include **unit mismatches** (Open3D uses meters, xArm SDK uses millimeters), rotation convention differences (quaternions vs. Euler angles vs. rotation matrices—choose one and convert consistently), and timestamp synchronization (ensure depth and color frames are temporally aligned, use `rs.align()` in pyrealsense2).

Calibration procedures and accuracy requirements

Hand-eye calibration determines the transformation between your camera and robot base, fundamentally enabling all coordinate transformations in your system. **Budget a full day for initial calibration** and expect to recalibrate periodically (monthly or after any physical changes to camera mounting).

The ArUco marker method provides the best balance of accuracy and ease. Print a 6×6 ArUco board with 50mm markers (use OpenCV's `aruco.GridBoard_create()` and print at actual size), mount it rigidly on a table within your workspace, and ensure it remains completely stationary throughout calibration. Move the robot through 15-20 diverse poses sampling different positions, orientations, and distances from the marker board. At each pose, record the robot's end-effector pose from forward kinematics and capture an image detecting the ArUco board pose relative to the camera.

Use OpenCV's `calibrateHandEye()` function with the Tsai method (generally most robust): `R_cam2base, t_cam2base = cv2.calibrateHandEye(R_gripper2base, t_gripper2base, R_target2cam, t_target2cam, method=cv2.CALIB_HAND_EYE_TSAI)`. The function solves the AX=XB equation relating gripper motion to camera observation changes.

Expected calibration accuracy should achieve **2-5mm translation error and 1-2° rotation error**. Validate by placing a test marker at a known position, detecting it with the camera, transforming to robot coordinates, and commanding the robot to that position. Measure the actual error with calipers or by placing a sharp pointer on the end effector. If errors exceed 5mm, recalibrate with more diverse poses or check for mechanical compliance in the camera mount.

Alternative calibration approaches include Easy HandEye (ROS package with interactive GUI), ChArUco boards (more robust detection), and sphere-based calibration using both RGB and depth. The JHU handeye_calib_camodocal package has proven reliable across many projects, recommending ~36 transforms for good calibration quality.

The RealSense D435 ships factory-calibrated and typically needs no additional depth calibration. If you observe systematic depth errors (e.g., all measurements 2% too short), use the Intel RealSense Viewer's "On-Chip Calibration" tool: point at a blank wall and run automatic calibration (15 seconds). Only recalibrate after drops, impacts, or obvious accuracy degradation.

Handling real-world depth sensing challenges

Depth camera noise represents the primary challenge for reliable grasping. The RealSense D435 uses active IR stereo, which produces **±5-10mm temporal jitter** and range-dependent noise increasing with distance. Flying pixels appear at object edges where IR patterns match incorrectly, while reflective surfaces (metal, glossy plastic) cause IR to reflect away producing missing depth data. Transparent materials like glass allow IR to pass through entirely, making them effectively invisible to depth cameras.

Combat these issues with a multi-stage filtering pipeline. Apply decimation to reduce resolution and computational load, spatial filtering (smooth_alpha=0.5, smooth_delta=20) to reduce noise while preserving edges, temporal filtering (smooth_alpha=0.4) to average across frames reducing jitter, and hole filling to interpolate missing data from valid neighbors. Process in this specific order for optimal results.

After point cloud generation, apply Open3D's statistical outlier removal to eliminate noise: `cl, ind = pcd.remove_statistical_outlier(nb_neighbors=20, std_ratio=2.0)`. This removes points whose average distance to nearest neighbors exceeds 2 standard deviations from the mean. You can also use radius outlier removal for cleaning up sparse noise.

Lighting optimization matters less for active IR depth cameras than RGB-based methods, but some guidelines apply. The D435 **performs better in bright ambient light** because it reduces sensor noise, though you should avoid direct sunlight on the IR sensors themselves. Consistent overhead diffused lighting proves ideal—avoid harsh shadows or spotlights creating high-contrast regions.

Material-specific challenges require different strategies. For **reflective objects** (metal utensils, glossy bottles), increase the laser power in RealSense settings, use longer exposure times (reduces frame rate), or apply matte coating for prototyping (hairspray or chalk spray create temporary matte surfaces for testing). In production, consider fusing multiple views from different angles or using known CAD models for shiny objects.

Transparent objects remain an unsolved research problem for single-view depth cameras. For month one, **simply exclude these from your test set**—avoid glass cups, clear plastic containers, and transparent medication bottles. Future solutions include matte backgrounds behind transparent objects (makes them partially visible), tactile sensing after initial contact, or known CAD model overlays when objects are predictable.

Dark objects (black plastic) absorb IR reducing signal strength. Use the High Accuracy preset in RealSense settings, increase exposure time, or add supplemental lighting. Most household objects in matte colors work reasonably well—test your specific objects early and identify any problematic items.

Object positioning uncertainty compounds depth noise. RealSense accuracy specifications state ±1% of distance, meaning 10mm error at 1m distance. **Always include 20mm safety margins** in grasp planning to account for cumulative uncertainties from depth sensing, calibration, robot positioning, and gripper mechanics.

Simulation options for testing before hardware deployment

PyBullet emerges as the optimal simulator for your 1-month timeline, offering **setup time under 1 hour**, fast performance (faster than real-time), Python-native API, built-in depth rendering matching RealSense characteristics, and low learning curve for teams familiar with Python. Major research projects (VGN, GraspNet) include PyBullet integration examples you can adapt directly.

Install with `pip install pybullet` and import pybullet_data for basic assets. Load your robot URDF (use UR5 as proxy for UFactory Lite 6 if official URDF unavailable), create a table and objects using primitive shapes or YCB dataset meshes, simulate a depth camera using `getCameraImage()` with appropriate view and projection matrices, and test grasp execution with physics validation before hardware.

A basic PyBullet setup requires 10-15 lines of Python code: connect to the visualizer, set gravity, load plane and robot URDFs, create objects with poses, step simulation, and render camera images. The API provides collision checking, force sensing, and joint control—everything needed to validate grasp poses before deploying to real hardware.

CoppeliaSim (formerly V-REP) represents a middle-ground option with more sophisticated visualization, better URDF support, extensive sensor models, and roughly 4 hours setup time. The Python API (`sim` module) enables scripting similar to PyBullet. Use CoppeliaSim if you need higher-fidelity sensor simulation or more realistic rendering for presentations.

Avoid Gazebo for your timeline despite its ROS integration advantages. Setup time reaches 4+ hours, performance is significantly slower than PyBullet, and the learning curve is steep for non-ROS users. Gazebo makes sense after migrating to ROS2 when you need standardized simulation infrastructure.

Isaac Sim from NVIDIA offers very fast GPU-accelerated physics and photorealistic rendering, but setup time spans 1-2 days and the system demands significant GPU resources (RTX 3080+). Consider Isaac Sim for future RL-based training where you need 50,000+ grasp attempts in simulation, but it provides diminishing returns for initial prototyping.

Reinforcement learning simulation is explicitly not recommended for month one. RL requires 50,000+ training episodes, days to weeks of training time even with GPU acceleration, complex sim-to-real transfer, and extensive hyperparameter tuning. The cost-benefit analysis strongly favors using pre-trained models (Dex-Net, AnyGrasp, GraspNet) that already solve the grasp planning problem with supervised or self-supervised learning on synthetic data.

Quick testing without full simulation includes offline visualization (render point clouds + gripper meshes in Open3D to verify grasp poses visually), recorded data testing (capture 50-100 test images with manual grasp labels to evaluate planner offline), and safe physical testing starting with foam objects, then cardboard, finally rigid objects—always at 10-20% speed initially.

Safety considerations for assistive robotics applications

Safety represents the paramount concern for assistive robotics serving ALS patients who may have limited ability to respond to emergencies. **Always have a caregiver present** during operation—patients may lack the motor control to press emergency stops. Design for **predictability over efficiency**: announce all actions verbally and visually, use slow deliberate movements rather than optimized speed, and provide clear abort options requiring minimal input (any key press, large touch targets, voice commands).

Implement multi-layer safety systems starting with a **hardware emergency stop**: large red mushroom button, hardwired to the xArm controller (not software-mediated), positioned within easy reach of both patient and caregiver, tested daily before operation. This represents your primary safety mechanism.

Add software backup safety monitoring: space bar or designated key press triggering emergency stop via xArm SDK's `set_state(4)` command, watchdog timer stopping motion if main loop hangs for over 500ms (indicates software crash), automatic timeout after 30 seconds of continuous motion (prevents runaway behavior), and force monitoring if sensors available (stop on unexpected resistance).

Configure speed and force limits following ISO 13482 recommendations for personal care robots: maximum contact force 50N transient (equivalent to firm handshake) or 25N sustained, maximum speed near human 250mm/s (about 10 inches/second—slow enough to react), workspace speed up to 500mm/s when clear of patient, and approach speed 100mm/s for final grasp (very deliberate for predictability). Set acceleration to 500mm/s² and jerk limit to 2000mm/s³ to ensure smooth motion without sudden jerks startling the patient.

Define and enforce workspace boundaries in three dimensions: X/Y/Z limits defining reachable space (-500 to 500mm typical), patient exclusion zone (cylindrical region around patient's head/body), no-go zones for obstacles (furniture, medical equipment), and vertical limits preventing downward collision with table or lap. Check boundaries before every motion command, rejecting any trajectory that would violate constraints.

Collision detection capabilities for the UFactory Lite 6 include built-in collision detection mode setting sensitivity (check documentation for your specific model), force/torque sensing if equipped (stop on unexpected force), and predictive collision checking using depth data (advanced—defer to month 2+). For month one, workspace boundaries and speed limits provide adequate safety with proper testing.

ALS-specific considerations learned from user studies include: 82% of patients rated object retrieval as "very relevant" making grasping a high-value task, users prioritize quality over speed ("we have practiced patience"), reliability builds trust—system should succeed consistently rather than fail occasionally even if faster, and patients need independence from requesting help repeatedly (key dignity concern). The JACO arm study found 93.3% user satisfaction with **71.4% daily usage**, suggesting assistive arms meeting user needs see high adoption.

Regulatory requirements: your month-one system qualifies as **research equipment, not a medical device**. IRB approval is required for any patient testing. Future deployment as a medical device requires FDA clearance (510(k) pathway likely, comparing to JACO) or CE marking in Europe—both require formal risk analysis (ISO 14971), design verification and validation, usability testing with target population, and clinical evaluation. Budget 6-12 months for regulatory approval when transitioning from research to clinical device.

Common pitfalls and implementation best practices

Coordinate frame confusion causes 50%+ of initial robotics implementation bugs. Every point exists in a specific coordinate frame: camera optical frame, camera housing frame, robot base frame, end-effector frame, world frame. Document your convention explicitly, visualize axes at each frame using Open3D or

RViz to verify transformations, use clear variable naming (`point_camera`, `point_base`, `T_base_camera`), and test transforms manually by placing markers at known positions and verifying computed results.

Units mismatch ranks second in common errors. Open3D and RealSense use meters, xArm SDK uses millimeters, some libraries use centimeters, angle measurements mix radians and degrees. Define unit constants at the top of your code: `MM_TO_M = 0.001`, `M_TO_MM = 1000.0`, add assertions checking reasonable ranges (`assert 0.2 < z_meters < 2.0`), and document units in function signatures and variable names (`gripper_width_mm`).

Skipping calibration or using "close enough" values produces catastrophic results. Hand-eye calibration errors propagate through your entire pipeline. Without proper calibration, expect **30% success rates versus 90% with correct calibration**. Budget the time for proper calibration, validate with test measurements, and recalibrate monthly or after any physical changes.

Ignoring gripper dimensions during planning causes frequent collisions. The grasp pose represents the tool center point (TCP), but the gripper has physical dimensions extending 50-150mm from TCP. Implement bounding box collision checking: create a box around the gripper, transform it to the grasp pose, check for intersection with the point cloud or scene geometry, and reject grasps causing collisions.

Not accounting for uncertainty treats depth measurements and robot positions as exact values. In reality, cumulative uncertainty from depth noise (±10mm), calibration error (2-5mm), robot repeatability (±0.1mm for Lite 6), and gripper mechanics (1-3mm) totals 15-20mm. Always include safety margins: approach 20mm above computed grasp point, open gripper 5-10mm wider than computed width, move slowly in final approach to detect contact.

Testing on hard objects first risks damage to gripper, object, or environment. Start with foam shapes that compress safely, progress to cardboard boxes and soft plastic bottles, finally attempt rigid objects (ceramic cups, metal utensils) only after successful soft testing. Test at 10-20% of normal speed initially, gradually increasing as confidence builds.

Full speed without validation represents a dangerous rookie mistake. Always start at reduced speed (50-100mm/s), verify motion paths visually before execution, test emergency stop at every speed setting, and only increase speed after extensive successful testing.

Trusting raw depth data without filtering produces noisy, unreliable point clouds. The RealSense D435 generates considerable noise without post-processing. Always apply the complete filter pipeline: decimation → spatial → temporal → hole filling, then statistical outlier removal on resulting point clouds.

Implementing algorithms from papers directly assuming paper conditions match reality leads to frustration. Papers assume perfect depth data, known object CAD models, simplified geometries, and controlled lighting. Start with simple heuristics (highest point + vertical approach), validate on real data, then gradually add sophistication based on actual failure modes observed.

No failure recovery causes systems to hang on errors. Wrap all robot commands in try-catch blocks, implement return-to-home on any error, provide manual recovery options, log all failures with point cloud snapshots for debugging, and design the state machine with explicit error states and recovery transitions.

Recommended resources and learning materials

Start immediately with these essential resources:

Official UFactory integration materials provide the fastest path to working code. The xArm-Python-SDK repository (github.com/xArm-Developer/xArm-Python-SDK) contains API documentation and control examples, while ufactory_vision (github.com/xArm-Developer/ufactory_vision) provides vision-based grasping demos specifically for Lite 6 with RealSense D435 support. The official Lite 6 Developer Manual (ufactory.cc documentation section) specifies kinematics, communication protocols, and safety specifications.

For grasp planning implementations, clone Contact-GraspNet (github.com/NVlabs/contact_graspnet) as your primary reference with PyTorch versions available for easier modification, GraspNet-1Billion baseline (github.com/graspnet/graspnet-baseline) providing well-documented training and inference code, and AnyGrasp SDK (github.com/graspnet/anygrasp_sdk) after license approval. The GPD repository (github.com/atenpas/gpd) serves as excellent learning material for understanding point cloud grasp detection even if you use learning-based methods ultimately.

Hand-eye calibration tutorials prove critical for achieving accuracy. Follow MoveIt's hand-eye calibration tutorial (moveit.picknik.ai documentation) for step-by-step ROS integration, use Easy HandEye (github.com/IFL-CAMP/easy_handeye) for automated calibration with interactive GUI, or implement the JHU handeye_calib_camodocal (github.com/jhu-lcsr/handeye_calib_camodocal) proven reliable method requiring ~36 transforms for good results.

Academic foundations and theory:

MIT's "Manipulation: Perception, Planning and Control" course at manipulation.csail.mit.edu provides production-ready code examples for grasp planning, excellent point cloud processing tutorials, and geometric grasping implementation. Northwestern's Modern Robotics (modernrobotics.northwestern.edu)

offers comprehensive theory on grasping and manipulation with video lectures on wrench spaces, force closure, and contact kinematics.

Key research papers include Contact-GraspNet (ICRA 2021, arXiv:2103.14127) for contact-based representation in cluttered scenes, AnyGrasp (IEEE T-RO 2023, arXiv:2212.08333) achieving 93.3% success with temporal consistency, GraspNet-1Billion (CVPR 2020) establishing the large-scale benchmark and dataset, GPD (IJRR 2017, arXiv:1706.09911) for classical point cloud grasp detection, and the 2023 survey "Deep Learning Approaches to Grasp Synthesis" (IEEE TRO) reviewing 150+ papers comprehensively.

Community support and troubleshooting:

Join the Open Robotics Discord (discord.com/servers/open-robotics-1077825543698927656) for official ROS/Gazebo support, reddit.com/r/robotics with 500k+ members mixing hobbyists and professionals, and ROS Discourse (discourse.ros.org) for tagged questions on perception and manipulation. Search Stack Overflow using tags [ros] + [moveit] + [grasping] for common integration issues, [point-cloud-library] for PCL-specific problems, and [realsense] for camera troubleshooting.

Assistive robotics user needs:

Study the German ALS user needs assessment (Neurological Research and Practice 2024, DOI:10.1186/s42466-024-00342-3) where 85 patients shared expectations and 14 users reported experiences. Harvard's soft robotic wearable for ALS (Science Translational Medicine 2023) demonstrates design principles for ALS-specific adaptations. The tetraplegia robotic arm needs assessment (Journal of NeuroEngineering and Rehabilitation 2025, DOI:10.1186/s12984-025-01642-8) emphasizes quality over speed and handling repetitive tasks.

Quick reference repositories:

The Tabletop HandyBot project (intelrealsense.com/building-an-ai-powered-robotic-arm) provides complete end-to-end reference architecture using RealSense D435 + AI-powered grasping. The "Robotic Grasping Papers" collection (github.com/rhett-chen/Robotic-grasping-papers) categorizes recent papers by approach type with regular 2024-2025 updates. GraspIt! simulator (graspit-simulator.github.io) enables testing grasp quality metrics and visualizing wrench spaces before hardware deployment.

Final recommendations and success criteria

For immediate action this week: Apply for the AnyGrasp SDK license today (anticipate 2-3 day approval), clone and set up GraspNet-1Billion baseline as your backup option, install Open3D, pyrealsense2, and Robotics Toolbox to establish the development environment, review the MIT manipulation course materials for foundational understanding, and test your RealSense D435 with basic point cloud capture to verify hardware functionality.

Architectural decision: Proceed with Python-native architecture using Open3D for perception, AnyGrasp (primary) or GraspNet-1Billion (backup) for grasp planning, Robotics Toolbox or xArm SDK IK for motion planning, and PyBullet for simulation testing. This represents the only approach achieving a working prototype in your 1-month timeline while leveraging your team's Python/AI expertise and mechanical engineering knowledge without requiring ROS experience.

Critical success factors that determine project outcome include accurate hand-eye calibration (budget full day, validate to 2-5mm), starting with simple objects and gradually increasing difficulty, comprehensive safety systems implemented and tested before patient interaction, and focusing on reliability over sophistication—a simple system that works beats a complex system that fails.

Month one completion criteria define success as: camera captures point cloud and RF-DETR segments objects correctly (perception pipeline functional), system generates ranked grasp poses for target objects with quality scores (planning pipeline operational), arm safely executes grasps on soft test objects with E-stop and workspace limits verified (control pipeline safe and operational), GUI provides object selection, status display, and emergency stop capability (user interface functional), and documentation describes architecture, calibration procedures, and failure recovery (knowledge captured for iteration).

Achieve 70%+ grasp success on 10 representative household objects (cups, bottles, utensils, phone, medication containers) with safe, predictable motion as your quantitative success metric. This success rate on a curated test set demonstrates working technology ready for refinement, provides foundation for user studies and feedback collection, and justifies investment in additional development time.

When to escalate or pivot: If calibration repeatedly fails to achieve 5mm accuracy, check camera mount rigidity and ensure robot positioning repeatability. If grasp detection produces no reasonable candidates, verify point cloud quality and coordinate frame transforms. If the system cannot achieve 50%+ success on simple objects by week 3, reassess grasp planner choice or simplify to purely heuristic approaches (top-down grasps only). The 1-month timeline is aggressive but achievable with focus—avoid scope creep adding features beyond the core pipeline.

Migration to ROS2 becomes justified when you need sophisticated collision-aware motion planning beyond basic IK, want to integrate multiple sensors or cameras, require comprehensive simulation infrastructure for testing, or begin scaling beyond single-arm applications. Plan this migration for months 3-6 after demonstrating core functionality and team members learning ROS2 fundamentals through tutorials. The Python-native architecture with clear module boundaries enables gradual migration minimizing disruption.

This assistive robotics system for ALS patients represents meaningful work with direct human impact. **Prioritize user safety, reliability, and dignity** throughout development. The 1-month timeline targets a functional prototype demonstrating feasibility—subsequent months will refine reliability, add capabilities, and incorporate user feedback toward a robust assistive device improving patient independence and quality of life.

References

1. Ten Pas, A., Gualtieri, M., Saenko, K., & Platt, R. (2017). Grasp Pose Detection in Point Clouds. *arXiv preprint arXiv:1706.09911*. Available: https://arxiv.org/abs/1706.09911

2. Fang, H., Wang, C., Fang, H., & Lu, C. (2023). AnyGrasp: Robust and Efficient Grasp Perception in Spatial and Temporal Domains. *IEEE Transactions on Robotics*. Available: https://ieeexplore.ieee.org/document/10167687/

3. Sundermeyer, M., Mousavian, A., Triebel, R., & Fox, D. (2021). Contact-GraspNet: Efficient 6-DoF Grasp Generation in Cluttered Scenes. *arXiv preprint arXiv:2103.14127*. Available: https://arxiv.org/abs/2103.14127

4. Fang, H. S., Wang, C., Gou, M., & Lu, C. (2020). GraspNet-1Billion: A Large-Scale Benchmark for General Object Grasping. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition* (CVPR 2020). Available: https://graspnet.net/

5. Mahler, J., Liang, J., Niyaz, S., Laskey, M., Doan, R., Liu, X., Ojea, J. A., & Goldberg, K. (2017). Dex-Net 2.0: Deep Learning to Plan Robust Grasps with Synthetic Point Clouds and Analytic Grasp Metrics. *arXiv preprint arXiv:1703.09312*. Available: https://arxiv.org/abs/1703.09312

6. Open3D Development Team. (2024). RealSense with Open3D. *Open3D Documentation*. Available: https://www.open3d.org/docs/release/tutorial/sensor/realsense.html

7. UFactory. (2024). xArm ROS2 Developer Packages. *GitHub Repository*. Available: https://github.com/xArm-Developer/xarm_ros2

8. Tedrake, R. (2024). Bin Picking. In *Robotic Manipulation: Perception, Planning, and Control*. MIT CSAIL. Available: https://manipulation.csail.mit.edu/clutter.html

9. Intel RealSense. (2024). Building an AI-powered robotic arm assistant using D435. Available: https://www.intelrealsense.com/building-an-ai-powered-robotic-arm-assistant-using-d435/

10. Intel RealSense. (2024). Post-processing filters. *RealSense Developer Documentation*. Available: https://dev.intelrealsense.com/docs/post-processing-filters

11. AnyGrasp SDK. (2024). *GitHub Repository*. Available: https://github.com/graspnet/anygrasp_sdk

12. O'Donnell, K., Duvall, J., Lobo-Prat, J., et al. (2023). Restoring arm function with a soft robotic wearable for individuals with amyotrophic lateral sclerosis. *Science Translational Medicine*, 15(714). DOI: 10.1126/scitranslmed.add1504

13. Standard Bots. (2024). Collaborative robot safety standards you must know. Available: https://standardbots.com/blog/collaborative-robot-safety-standards

14. NVLabs. (2021). Contact-GraspNet: Efficient 6-DoF Grasp Generation. *GitHub Repository*. Available: https://github.com/NVlabs/contact_graspnet

15. ETH Zurich ASL. (2024). VGN: Real-time 6 DOF grasp detection in clutter. *GitHub Repository*. Available: https://github.com/ethz-asl/vgn

16. MathWorks. (2024). What Is Robot Hand-Eye Calibration? *MATLAB & Simulink Documentation*. Available: https://www.mathworks.com/help/vision/ug/what-is-robot-hand-eye-calibration.html

17. Borgan, Ø. (2023). The practical guide to 3D hand-eye calibration. *Medium/Zivid*. Available: https://medium.com/zivid/the-practical-guide-to-3d-hand-eye-calibration-3c29c0148f62

18. Chen, R. (2024). Robotic Grasping Papers. *GitHub Repository*. Available: https://github.com/rhett-chen/Robotic-grasping-papers

19. Du, G. (2024). Vision-based Robotic Grasping. *GitHub Repository*. Available: https://github.com/GeorgeDu/vision-based-robotic-grasping

20. MoveIt Documentation. (2024). Grasping using Deep Learning. Available: https://moveit.ros.org/deep%20learning/grasping/moveit/3d%20perception/2020/09/28/grasp-deep-learning.html

---

Report generated: November 2025

For the Access-Ability-Arm Project

GitHub: https://github.com/Access-Ability-Arm/access-ability-arm