14 Low Level Design

2025-10-19

Contents

1 Low-Level Design (LLD)

1.1 Vision-Based Pick and Place Robotic System

Document Version: 1.0 **Last Updated:** 2025-10-18 **Status:** Complete

---

1.2 Table of Contents

1.3 1. Introduction

1.3.1 1.1 Purpose

This Low-Level Design (LLD) document provides detailed technical specifications for implementing the Vision-Based Pick and Place Robotic System. It translates the High-Level Design (HLD) into concrete component designs, data structures, algorithms, and interfaces suitable for direct implementation.

1.3.2 1.2 Scope

This document covers: - Detailed class/component designs for all subsystems - Data structures and algorithms - ROS2 node specifications (topics, services, actions, parameters) - Database schemas - REST/gRPC API specifications - State machine implementations - Error handling strategies - Performance optimization techniques

1.3.3 1.3 Design Principles

Principle	Implementation
Modularity	Each subsystem is a separate ROS2 package with defined interfaces
Reusability	Common utilities in shared libraries (e.g., `robot_utils`)
Testability	Dependency injection, mock interfaces, unit test hooks
Scalability	Stateless services, horizontal scaling via Docker/K8s
Maintainability	Clear naming, comprehensive logging, configuration files
Real-Time	FIFO scheduling, memory pre-allocation, bounded execution time

1.3.4 1.4 Notation Conventions

- **Class Diagrams:** Simplified UML notation (ASCII art)

- **Sequence Diagrams:** Time flows top-to-bottom
- **Data Types:** C++ types (e.g., `std::vector<double>`, `geometry_msgs::msg::Pose`)
- **Units:** SI units (meters, radians, Newtons) unless specified

---

1.4 2. Component-Level Design

1.4.1 2.1 System Decomposition

```
visionpickplace_ws/
  src/
     vision_pipeline/          # Vision subsystem
        camera_driver/
        image_processor/
        object_detector/
        pose_estimator/
     grasp_planning/           # Grasp planning subsystem
        grasp_synthesizer/
        grasp_evaluator/
     motion_planning/          # Motion planning subsystem (MoveIt2 configs)
        moveit_config/
        collision_objects/
     control/                  # Control subsystem
        controllers/
        hardware_interface/
     gripper_control/          # Gripper subsystem
        robotiq_driver/
     task_orchestration/       # Orchestration subsystem
        task_manager/
        behavior_trees/
     monitoring/               # Monitoring subsystem
        metrics_collector/
        alerting/
     common/                   # Shared libraries
        robot_msgs/          # Custom ROS2 messages
        robot_utils/         # Utility functions
        robot_interfaces/    # Service/action definitions
     web_dashboard/           # Web UI
         backend/             # FastAPI server
         frontend/            # React app
```

1.4.2 2.2 Technology Mapping

Component	Language	Framework/Library	Deployment
Vision Pipeline	C++ / Python	ROS2, OpenCV, PyTorch	Docker (GPU)
Grasp Planning	Python	NumPy, SciPy, Open3D	Docker (GPU)
Motion Planning	C++	MoveIt2, OMPL	Docker

Component	Language	Framework/Library	Deployment
Control	C++	ros2_control, RT-Linux	Bare metal / Docker
Orchestration	C++ / Python	BehaviorTree.CPP	Docker
Monitoring	Python	Prometheus, InfluxDB	Docker
Web Backend	Python	FastAPI, SQLAlchemy	Docker
Web Frontend	TypeScript	React, Next.js	Docker (Nginx)

1.5 3. Vision Subsystem

1.5.1 3.1 Architecture

```
Camera Driver      > /camera/color/image_raw (sensor_msgs/Image)
             /camera/depth/image_rect (sensor_msgs/Image)
                     /camera/color/camera_info (sensor_msgs/CameraInfo)


Image Processor    > /vision/preprocessed/image (sensor_msgs/Image)



Object Detector    > /vision/detections (vision_msgs/Detection2DArray)



Pose Estimator     > /vision/object_poses (geometry_msgs/PoseArray)
```

1.5.2 3.2 Camera Driver Node

Package: `camera_driver` **Node Name:** `realsense_driver_node` **Executable:** `realsense_driver`

```
class RealsenseDriver : public rclcpp::Node {
public:
    RealsenseDriver(const rclcpp::NodeOptions& options);
    ~RealsenseDriver();

private:
    // Camera interface
    rs2::pipeline pipeline_;
    rs2::config config_;

    // Publishers
```

```cpp
    image_transport::Publisher color_pub_;
    image_transport::Publisher depth_pub_;
    rclcpp::Publisher<sensor_msgs::msg::CameraInfo>::SharedPtr camera_info_pub_;

    // Timers
    rclcpp::TimerBase::SharedPtr capture_timer_;

    // Parameters
    int width_;          // Default: 1280
    int height_;         // Default: 720
    int fps_;            // Default: 30
    std::string serial_; // Camera serial number

    // Methods
    void captureAndPublish();
    void loadParameters();
    sensor_msgs::msg::Image convertToRosImage(const rs2::frame& frame);
    sensor_msgs::msg::CameraInfo getCameraInfo(const rs2::stream_profile& profile);
};
```

1.5.2.1 3.2.1 Class Design

```yaml
# camera_driver/config/realsense.yaml
realsense_driver:
  ros__parameters:
    width: 1280
    height: 720
    fps: 30
    enable_color: true
    enable_depth: true
    enable_infra: false
    align_to_color: true
    depth_min_range: 0.3   # meters
    depth_max_range: 3.0   # meters
    serial_number: ""      # Auto-detect if empty
    frame_id: "camera_color_optical_frame"
    publish_rate: 30.0     # Hz
```

1.5.2.2 3.2.2 Configuration Parameters

1.5.2.3 3.2.3 Published Topics

Topic	Type	Frequency	QoS
`/camera/color/image_raw`	`sensor_msgs/msg/Image`	30 Hz	Best Effort
`/camera/depth/image_rect_raw`	`sensor_msgs/msg/Image`	30 Hz	Best Effort
`/camera/color/camera_info`	`sensor_msgs/msg/CameraInfo`	30 Hz	Best Effort

Topic	Type	Frequency	QoS
`/camera/depth/camera_info`	`sensor_msgs/msg/CameraInfo`	30 Hz	Best Effort

1.5.3 3.3 Object Detector Node

Package: `object_detector` **Node Name:** `yolo_detector_node` **Executable:** `yolo_detector`

```python
class YoloDetector(Node):
    def __init__(self):
        super().__init__('yolo_detector_node')

        # Model
        self.model = None  # YOLO model (PyTorch)
        self.device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

        # Subscribers
        self.image_sub = self.create_subscription(
            Image, '/vision/preprocessed/image', self.image_callback, 10)

        # Publishers
        self.detections_pub = self.create_publisher(
            Detection2DArray, '/vision/detections', 10)
        self.debug_image_pub = self.create_publisher(
            Image, '/vision/debug/detections', 10)

        # Parameters
        self.declare_parameter('model_path', '/models/yolov8n.pt')
        self.declare_parameter('confidence_threshold', 0.5)
        self.declare_parameter('iou_threshold', 0.45)
        self.declare_parameter('max_detections', 10)

        # TensorRT engine (optional)
        self.use_tensorrt = self.declare_parameter('use_tensorrt', False).value

        self.load_model()

    def load_model(self):
        """Load YOLO model (PyTorch or TensorRT)."""
        model_path = self.get_parameter('model_path').value
        if self.use_tensorrt:
            self.model = self.load_tensorrt_model(model_path)
        else:
            self.model = torch.hub.load('ultralytics/yolov8', 'custom', path=model_path)
            self.model.to(self.device)
            self.model.eval()
```

```python
    def image_callback(self, msg: Image):
        """Process incoming image and detect objects."""
        # Convert ROS Image to NumPy/Torch tensor
        cv_image = self.bridge.imgmsg_to_cv2(msg, desired_encoding='bgr8')

        # Run inference
        with torch.no_grad():
            results = self.model(cv_image)

        # Parse results
        detections = self.parse_detections(results)

        # Publish
        self.detections_pub.publish(detections)

        # Publish debug image
        if self.debug_image_pub.get_subscription_count() > 0:
            debug_img = self.draw_detections(cv_image, detections)
            self.debug_image_pub.publish(self.bridge.cv2_to_imgmsg(debug_img, 'bgr8'))

    def parse_detections(self, results) -> Detection2DArray:
        """Convert YOLO results to ROS Detection2DArray."""
        detections = Detection2DArray()
        detections.header.stamp = self.get_clock().now().to_msg()
        detections.header.frame_id = "camera_color_optical_frame"

        conf_thresh = self.get_parameter('confidence_threshold').value

        for detection in results.xyxy[0]:  # [x1, y1, x2, y2, conf, class]
            if detection[4] < conf_thresh:
                continue

            det = Detection2D()
            det.bbox.center.x = (detection[0] + detection[2]) / 2.0
            det.bbox.center.y = (detection[1] + detection[3]) / 2.0
            det.bbox.size_x = detection[2] - detection[0]
            det.bbox.size_y = detection[3] - detection[1]

            hypothesis = ObjectHypothesisWithPose()
            hypothesis.id = str(int(detection[5]))
            hypothesis.score = float(detection[4])
            det.results.append(hypothesis)

            detections.detections.append(det)

        return detections
```

1.5.3.1 3.3.1 Class Design

1.5.3.2 3.3.2 Performance Optimization

Optimization	Implementation	Speedup
TensorRT	Convert PyTorch model to TensorRT engine	3-5x faster
FP16 Precision	Use half-precision (16-bit) inference	2x faster
Batch Processing	Process multiple images in parallel	1.5x throughput
NMS Optimization	GPU-accelerated Non-Max Suppression	1.2x faster
Input Resizing	Resize to 640×640 (vs 1280×720)	2x faster

Target Latency: <50ms per frame (on Jetson Xavier NX with TensorRT FP16)

1.5.4 3.4 Pose Estimator Node

Package: `pose_estimator` **Node Name:** `pose_estimator_node` **Executable:** `pose_estimator`

```
class PoseEstimator(Node):
    def __init__(self):
        super().__init__('pose_estimator_node')

        # Subscribers
        self.detections_sub = message_filters.Subscriber(
            self, Detection2DArray, '/vision/detections')
        self.depth_sub = message_filters.Subscriber(
            self, Image, '/camera/depth/image_rect_raw')
        self.camera_info_sub = message_filters.Subscriber(
            self, CameraInfo, '/camera/depth/camera_info')

        # Synchronizer
        self.sync = message_filters.ApproximateTimeSynchronizer(
            [self.detections_sub, self.depth_sub, self.camera_info_sub],
            queue_size=10, slop=0.1)
        self.sync.registerCallback(self.callback)

        # Publishers
        self.poses_pub = self.create_publisher(
            PoseArray, '/vision/object_poses', 10)
        self.pointcloud_pub = self.create_publisher(
            PointCloud2, '/vision/object_pointclouds', 10)

        # TF2 broadcaster
        self.tf_broadcaster = tf2_ros.TransformBroadcaster(self)
```

```
        # Camera intrinsics
        self.K = None  # 3x3 intrinsic matrix
        self.D = None  # Distortion coefficients

        # PCA-based pose estimator
        self.pca_estimator = PCABasedPoseEstimator()

    def callback(self, detections_msg, depth_msg, camera_info_msg):
        """Estimate 6DoF pose for each detected object."""
        # Update camera intrinsics
        if self.K is None:
            self.K = np.array(camera_info_msg.k).reshape(3, 3)
            self.D = np.array(camera_info_msg.d)

        # Convert depth image
        depth_image = self.bridge.imgmsg_to_cv2(depth_msg, desired_encoding='16UC1')
        depth_image = depth_image.astype(np.float32) / 1000.0  # mm to meters

        poses = PoseArray()
        poses.header = detections_msg.header

        for det in detections_msg.detections:
            # Extract bounding box
            x_min = int(det.bbox.center.x - det.bbox.size_x / 2)
            y_min = int(det.bbox.center.y - det.bbox.size_y / 2)
            x_max = int(det.bbox.center.x + det.bbox.size_x / 2)
            y_max = int(det.bbox.center.y + det.bbox.size_y / 2)

            # Extract point cloud for object
            obj_cloud = self.extract_pointcloud(depth_image, x_min, y_min, x_max, y_max)

            # Estimate pose using PCA
            pose = self.pca_estimator.estimate(obj_cloud)

            poses.poses.append(pose)

            # Broadcast TF
            self.broadcast_transform(pose, det.results[0].id)

        self.poses_pub.publish(poses)

    def extract_pointcloud(self, depth_image, x_min, y_min, x_max, y_max):
        """Convert depth ROI to 3D point cloud."""
        points = []
        for v in range(y_min, y_max):
            for u in range(x_min, x_max):
                z = depth_image[v, u]
```

```python
                if z == 0 or np.isnan(z) or z > 3.0:  # Invalid depth
                    continue
                # Backproject to 3D
                x = (u - self.K[0, 2]) * z / self.K[0, 0]
                y = (v - self.K[1, 2]) * z / self.K[1, 1]
                points.append([x, y, z])
        return np.array(points)

    def broadcast_transform(self, pose, object_id):
        """Broadcast TF for object."""
        t = TransformStamped()
        t.header.stamp = self.get_clock().now().to_msg()
        t.header.frame_id = "camera_color_optical_frame"
        t.child_frame_id = f"object_{object_id}"
        t.transform.translation.x = pose.position.x
        t.transform.translation.y = pose.position.y
        t.transform.translation.z = pose.position.z
        t.transform.rotation = pose.orientation
        self.tf_broadcaster.sendTransform(t)
```

1.5.4.1 3.4.1 Class Design

```python
class PCABasedPoseEstimator:
    """Estimate object pose using Principal Component Analysis."""

    def estimate(self, points: np.ndarray) -> Pose:
        """
        Estimate 6DoF pose from 3D point cloud.

        Args:
            points: Nx3 array of 3D points

        Returns:
            Pose with position (centroid) and orientation (PCA axes)
        """
        if len(points) < 10:
            raise ValueError("Insufficient points for PCA")

        # Compute centroid
        centroid = np.mean(points, axis=0)

        # Center points
        centered = points - centroid

        # Compute covariance matrix
        cov = np.cov(centered.T)
```

```python
        # Eigen decomposition
        eigenvalues, eigenvectors = np.linalg.eig(cov)

        # Sort by eigenvalues (descending)
        idx = eigenvalues.argsort()[::-1]
        eigenvectors = eigenvectors[:, idx]

        # Ensure right-handed coordinate system
        if np.linalg.det(eigenvectors) < 0:
            eigenvectors[:, 2] *= -1

        # Convert rotation matrix to quaternion
        rotation_matrix = eigenvectors
        quaternion = self.rotation_matrix_to_quaternion(rotation_matrix)

        # Create Pose message
        pose = Pose()
        pose.position.x = centroid[0]
        pose.position.y = centroid[1]
        pose.position.z = centroid[2]
        pose.orientation.x = quaternion[0]
        pose.orientation.y = quaternion[1]
        pose.orientation.z = quaternion[2]
        pose.orientation.w = quaternion[3]

        return pose

    @staticmethod
    def rotation_matrix_to_quaternion(R: np.ndarray) -> np.ndarray:
        """Convert 3x3 rotation matrix to quaternion [x, y, z, w]."""
        trace = np.trace(R)
        if trace > 0:
            s = 0.5 / np.sqrt(trace + 1.0)
            w = 0.25 / s
            x = (R[2, 1] - R[1, 2]) * s
            y = (R[0, 2] - R[2, 0]) * s
            z = (R[1, 0] - R[0, 1]) * s
        else:
            # ... (full implementation omitted for brevity)
            pass
        return np.array([x, y, z, w])
```

1.5.4.2 3.4.2 PCA-Based Pose Estimation Algorithm

1.6 4. Grasp Planning Subsystem

1.6.1 4.1 Architecture

```
Grasp Synthesizer      > Generates candidate grasps



Grasp Evaluator        > Scores and ranks grasps



Collision Checker      > Validates grasp feasibility
```

1.6.2 4.2 Grasp Synthesizer Service

Package: `grasp_synthesizer` **Service Name:** `compute_grasps` **Service Type:** `robot_interfaces/srv/Compu`

1.6.2.1 4.2.1 Service Definition

```
# robot_interfaces/srv/ComputeGrasps.srv
# Request
geometry_msgs/Pose object_pose
float32 object_width   # meters
float32 object_depth   # meters
float32 object_height  # meters
string object_shape    # "box" | "cylinder" | "sphere" | "unknown"
---
# Response
Grasp[] grasps
bool success
string message
```

```
# robot_interfaces/msg/Grasp.msg
geometry_msgs/PoseStamped pose       # Grasp pose (TCP frame)
float32 quality                      # Grasp quality score [0, 1]
float32 approach_distance            # Pre-grasp offset (meters)
geometry_msgs/Vector3 approach_direction
float32 gripper_width                # Target gripper opening (meters)
float32 force                        # Target gripping force (Newtons)
```

```python
class GraspSynthesizer(Node):
    def __init__(self):
        super().__init__('grasp_synthesizer')
```

```python
        # Service
        self.srv = self.create_service(
            ComputeGrasps, 'compute_grasps', self.compute_grasps_callback)

        # Parameters
        self.declare_parameter('num_candidates', 20)
        self.declare_parameter('gripper_max_width', 0.085)  # Robotiq 2F-85
        self.declare_parameter('gripper_min_width', 0.0)
        self.declare_parameter('friction_coefficient', 0.5)

    def compute_grasps_callback(self, request, response):
        """Generate antipodal grasp candidates."""
        object_pose = request.object_pose
        shape = request.object_shape

        if shape == "box":
            grasps = self.generate_box_grasps(request)
        elif shape == "cylinder":
            grasps = self.generate_cylinder_grasps(request)
        else:
            grasps = self.generate_generic_grasps(request)

        # Sort by quality
        grasps.sort(key=lambda g: g.quality, reverse=True)

        # Return top candidates
        num_candidates = self.get_parameter('num_candidates').value
        response.grasps = grasps[:num_candidates]
        response.success = len(response.grasps) > 0
        response.message = f"Generated {len(response.grasps)} grasps"

        return response

    def generate_box_grasps(self, request) -> List[Grasp]:
        """Generate grasps for box-shaped objects."""
        grasps = []
        width = request.object_width
        depth = request.object_depth
        height = request.object_height

        # Grasp from top (along Z-axis)
        for angle in np.linspace(0, 2*np.pi, 8, endpoint=False):
            grasp = Grasp()
            grasp.pose.header.frame_id = "base_link"
            grasp.pose.pose = self.compute_grasp_pose(
                request.object_pose, approach="top", rotation=angle)
            grasp.quality = self.evaluate_grasp_quality(grasp, request)
```

```python
            grasp.approach_distance = 0.1  # 10 cm pre-grasp
            grasp.gripper_width = min(max(width, depth), self.get_parameter('gripper_max_width
            grasp.force = 20.0  # Newtons
            grasps.append(grasp)

        # Grasp from sides
        # ... (additional grasp generation logic)

        return grasps

    def evaluate_grasp_quality(self, grasp: Grasp, request) -> float:
        """
        Compute grasp quality metric (Ferrari-Canny metric).

        Quality = min_wrench / object_weight
        where min_wrench is the minimum wrench that can be resisted.
        """
        # Simplified quality metric based on geometry
        # Full implementation would use force closure analysis

        gripper_width = grasp.gripper_width
        object_width = request.object_width

        # Penalize if object is too wide for gripper
        if gripper_width > self.get_parameter('gripper_max_width').value:
            return 0.0

        # Higher quality for centered grasps
        width_ratio = gripper_width / self.get_parameter('gripper_max_width').value
        quality = 1.0 - abs(width_ratio - 0.5) * 0.5

        return max(0.0, min(1.0, quality))
```

1.6.2.2 4.2.2 Class Design

1.6.3 4.3 Collision Checker

Package: `grasp_evaluator` **Node Name:** `collision_checker_node`

```cpp
class CollisionChecker {
public:
    CollisionChecker(const planning_scene::PlanningScenePtr& scene)
        : planning_scene_(scene) {}

    bool isGraspValid(const Grasp& grasp) {
        // Set robot state to grasp pose
        robot_state::RobotState& state = planning_scene_->getCurrentStateNonConst();
```

```cpp
        // Convert grasp pose to joint positions (IK)
        const robot_model::JointModelGroup* jmg =
            state.getJointModelGroup("manipulator");

        bool ik_success = state.setFromIK(jmg, grasp.pose.pose);
        if (!ik_success) {
            return false;  // Unreachable
        }

        // Check self-collision
        collision_detection::CollisionRequest req;
        collision_detection::CollisionResult res;
        planning_scene_->checkSelfCollision(req, res, state);
        if (res.collision) {
            return false;
        }

        // Check environment collision
        planning_scene_->checkCollision(req, res, state);
        if (res.collision) {
            return false;
        }

        return true;
    }

private:
    planning_scene::PlanningScenePtr planning_scene_;
};
```

1.7 5. Motion Planning Subsystem

1.7.1 5.1 MoveIt2 Configuration

Package: `moveit_config` **Files:** - `config/ur5e.srdf` - Semantic Robot Description Format - `config/ompl_planning.yaml` - OMPL planner configuration - `config/kinematics.yaml` - IK solver configuration - `config/joint_limits.yaml` - Joint velocity/acceleration limits

```yaml
# config/ompl_planning.yaml
planning_plugin: ompl_interface/OMPLPlanner
request_adapters: >-
  default_planner_request_adapters/AddTimeOptimalParameterization
  default_planner_request_adapters/FixWorkspaceBounds
  default_planner_request_adapters/FixStartStateBounds
  default_planner_request_adapters/FixStartStateCollision
```

```yaml
  default_planner_request_adapters/FixStartStatePathConstraints

start_state_max_bounds_error: 0.1

planner_configs:
  RRTConnect:
    type: geometric::RRTConnect
    range: 0.0  # Auto-detect

  RRTstar:
    type: geometric::RRTstar
    range: 0.0
    goal_bias: 0.05
    rewire_factor: 1.1

  PRM:
    type: geometric::PRM
    max_nearest_neighbors: 10

manipulator:
  default_planner_config: RRTConnect
  planner_configs:
    - RRTConnect
    - RRTstar
    - PRM
  projection_evaluator: joints(shoulder_pan_joint,shoulder_lift_joint,elbow_joint)
  longest_valid_segment_fraction: 0.005
```

1.7.1.1 5.1.1 OMPL Planner Configuration

```yaml
# config/kinematics.yaml
manipulator:
  kinematics_solver: kdl_kinematics_plugin/KDLKinematicsPlugin
  kinematics_solver_search_resolution: 0.005
  kinematics_solver_timeout: 0.05
  kinematics_solver_attempts: 3
```

1.7.1.2 5.1.2 Kinematics Configuration

1.7.2 5.2 Motion Planning Action Server

Package: `motion_planning` **Action Name:** `/plan_and_execute` **Action Type:** `moveit_msgs/action/MoveGro`

1.7.2.1 5.2.1 Custom Action Definition

```
# robot_interfaces/action/PickPlace.action
# Goal
```

```
geometry_msgs/PoseStamped pick_pose
geometry_msgs/PoseStamped place_pose
float32 approach_distance  # Pre-grasp offset
float32 retreat_distance   # Post-grasp lift
---
# Result
bool success
string message
moveit_msgs/RobotTrajectory executed_trajectory
float32 execution_time
---
# Feedback
string status  # "planning_pick" | "executing_pick" | "planning_place" | "executing_place"
float32 progress  # 0.0 to 1.0
```

```cpp
class PickPlaceServer : public rclcpp::Node {
public:
    using PickPlace = robot_interfaces::action::PickPlace;
    using GoalHandle = rclcpp_action::ServerGoalHandle<PickPlace>;

    PickPlaceServer(const rclcpp::NodeOptions& options)
        : Node("pick_place_server", options) {

        // MoveIt interface
        move_group_ = std::make_shared<moveit::planning_interface::MoveGroupInterface>(
            shared_from_this(), "manipulator");

        // Action server
        action_server_ = rclcpp_action::create_server<PickPlace>(
            this, "pick_place",
            std::bind(&PickPlaceServer::handle_goal, this, _1, _2),
            std::bind(&PickPlaceServer::handle_cancel, this, _1),
            std::bind(&PickPlaceServer::handle_accepted, this, _1));
    }

private:
    void execute(const std::shared_ptr<GoalHandle> goal_handle) {
        auto goal = goal_handle->get_goal();
        auto result = std::make_shared<PickPlace::Result>();

        // Feedback
        auto feedback = std::make_shared<PickPlace::Feedback>();

        // Step 1: Plan pick motion
        feedback->status = "planning_pick";
        feedback->progress = 0.1;
```

```cpp
        goal_handle->publish_feedback(feedback);

        moveit::planning_interface::MoveGroupInterface::Plan pick_plan;
        bool pick_success = planPick(goal->pick_pose, goal->approach_distance, pick_plan);

        if (!pick_success) {
            result->success = false;
            result->message = "Pick planning failed";
            goal_handle->abort(result);
            return;
        }

        // Step 2: Execute pick
        feedback->status = "executing_pick";
        feedback->progress = 0.3;
        goal_handle->publish_feedback(feedback);

        moveit::core::MoveItErrorCode exec_result = move_group_->execute(pick_plan);
        if (exec_result != moveit::core::MoveItErrorCode::SUCCESS) {
            result->success = false;
            result->message = "Pick execution failed";
            goal_handle->abort(result);
            return;
        }

        // Step 3: Close gripper (via action client to gripper controller)
        feedback->progress = 0.5;
        goal_handle->publish_feedback(feedback);
        closeGripper(goal->pick_pose);  // Blocking call

        // Step 4: Retreat
        feedback->progress = 0.6;
        retreat(goal->retreat_distance);

        // Step 5: Plan place motion
        feedback->status = "planning_place";
        feedback->progress = 0.7;
        goal_handle->publish_feedback(feedback);

        moveit::planning_interface::MoveGroupInterface::Plan place_plan;
        bool place_success = planPlace(goal->place_pose, place_plan);

        // Step 6: Execute place
        feedback->status = "executing_place";
        feedback->progress = 0.9;
        goal_handle->publish_feedback(feedback);
```

```cpp
        move_group_->execute(place_plan);

        // Step 7: Open gripper
        openGripper();

        // Success
        result->success = true;
        result->message = "Pick and place completed";
        result->execution_time = /* compute duration */;
        goal_handle->succeed(result);
    }

    bool planPick(const geometry_msgs::msg::PoseStamped& target,
                  double approach_distance,
                  moveit::planning_interface::MoveGroupInterface::Plan& plan) {
        // Compute pre-grasp pose (offset along approach direction)
        geometry_msgs::msg::PoseStamped pre_grasp = target;
        pre_grasp.pose.position.z += approach_distance;

        // Plan to pre-grasp
        move_group_->setPoseTarget(pre_grasp);
        bool success = (move_group_->plan(plan) == moveit::core::MoveItErrorCode::SUCCESS);

        if (!success) return false;

        // Append Cartesian path from pre-grasp to grasp
        std::vector<geometry_msgs::msg::Pose> waypoints = {target.pose};
        moveit_msgs::msg::RobotTrajectory cartesian_traj;
        double fraction = move_group_->computeCartesianPath(
            waypoints, 0.01 /* eef_step */, 0.0 /* jump_threshold */, cartesian_traj);

        if (fraction < 0.95) {
            RCLCPP_WARN(get_logger(), "Cartesian path only %.2f%% complete", fraction * 100);
            return false;
        }

        // Merge trajectories
        // ... (trajectory merging logic)

        return true;
    }

    moveit::planning_interface::MoveGroupInterface::SharedPtr move_group_;
    rclcpp_action::Server<PickPlace>::SharedPtr action_server_;
};
```

1.7.2.2 5.2.2 Pick and Place Server

1.8 6. Control Subsystem

1.8.1 6.1 ros2_control Configuration

Package: `control` **File:** `config/ur5e_controllers.yaml`

```yaml
controller_manager:
  ros__parameters:
    update_rate: 1000  # Hz (1 kHz control loop)

    joint_state_broadcaster:
      type: joint_state_broadcaster/JointStateBroadcaster

    joint_trajectory_controller:
      type: joint_trajectory_controller/JointTrajectoryController

    forward_position_controller:
      type: forward_command_controller/ForwardCommandController

joint_trajectory_controller:
  ros__parameters:
    joints:
      - shoulder_pan_joint
      - shoulder_lift_joint
      - elbow_joint
      - wrist_1_joint
      - wrist_2_joint
      - wrist_3_joint

    command_interfaces:
      - position

    state_interfaces:
      - position
      - velocity

    state_publish_rate: 100.0
    action_monitor_rate: 20.0

    allow_partial_joints_goal: false
    allow_integration_in_goal_trajectories: true

    constraints:
      stopped_velocity_tolerance: 0.01
      goal_time: 0.5
      shoulder_pan_joint:
        trajectory: 0.05
```

```
        goal: 0.01
      # ... (similar for other joints)
```

1.8.2 6.2 Hardware Interface

Package: `hardware_interface` **File:** `src/ur_hardware_interface.cpp`

```
class URHardwareInterface : public hardware_interface::SystemInterface {
public:
    hardware_interface::CallbackReturn on_init(
        const hardware_interface::HardwareInfo& info) override {

        // Store hardware info
        info_ = info;

        // Initialize joint states
        joint_positions_.resize(info_.joints.size(), 0.0);
        joint_velocities_.resize(info_.joints.size(), 0.0);
        joint_efforts_.resize(info_.joints.size(), 0.0);
        joint_position_commands_.resize(info_.joints.size(), 0.0);

        // Connect to robot (TCP/IP socket)
        robot_ip_ = info_.hardware_parameters["robot_ip"];
        robot_port_ = std::stoi(info_.hardware_parameters["robot_port"]);

        return hardware_interface::CallbackReturn::SUCCESS;
    }

    std::vector<hardware_interface::StateInterface> export_state_interfaces() override {
        std::vector<hardware_interface::StateInterface> state_interfaces;
        for (size_t i = 0; i < info_.joints.size(); ++i) {
            state_interfaces.emplace_back(
                info_.joints[i].name, hardware_interface::HW_IF_POSITION, &joint_positions_[i]
            state_interfaces.emplace_back(
                info_.joints[i].name, hardware_interface::HW_IF_VELOCITY, &joint_velocities_[i
            state_interfaces.emplace_back(
                info_.joints[i].name, hardware_interface::HW_IF_EFFORT, &joint_efforts_[i]);
        }
        return state_interfaces;
    }

    std::vector<hardware_interface::CommandInterface> export_command_interfaces() override {
        std::vector<hardware_interface::CommandInterface> command_interfaces;
        for (size_t i = 0; i < info_.joints.size(); ++i) {
            command_interfaces.emplace_back(
                info_.joints[i].name, hardware_interface::HW_IF_POSITION,
                &joint_position_commands_[i]);
        }
```

```
        return command_interfaces;
    }

    hardware_interface::return_type read(
        const rclcpp::Time& time, const rclcpp::Duration& period) override {

        // Read actual joint positions from robot
        ur_driver_->readJointStates(joint_positions_, joint_velocities_, joint_efforts_);

        return hardware_interface::return_type::OK;
    }

    hardware_interface::return_type write(
        const rclcpp::Time& time, const rclcpp::Duration& period) override {

        // Send joint position commands to robot
        ur_driver_->writeJointCommands(joint_position_commands_);

        return hardware_interface::return_type::OK;
    }

private:
    std::vector<double> joint_positions_;
    std::vector<double> joint_velocities_;
    std::vector<double> joint_efforts_;
    std::vector<double> joint_position_commands_;

    std::unique_ptr<URDriver> ur_driver_;
    std::string robot_ip_;
    int robot_port_;
};
```

1.9 7. Task Orchestration Subsystem

1.9.1 7.1 Behavior Tree Design

Package: `task_orchestration` **File:** `behavior_trees/pick_place.xml`

```
<root main_tree_to_execute="PickPlaceTree">
    <BehaviorTree ID="PickPlaceTree">
        <Sequence name="MainSequence">

            <!-- Initialize -->
            <Action ID="InitializeSystem" />

            <!-- Main loop -->
            <RepeatUntilFailure num_cycles="1000">
```

```xml
            <Sequence name="PickPlaceLoop">

                <!-- Wait for trigger -->
                <Condition ID="WaitForTrigger" />

                <!-- Scan workspace -->
                <Action ID="CaptureImage" />

                <!-- Detect objects -->
                <Action ID="DetectObjects" output="{detections}" />

                <!-- Check if objects found -->
                <Condition ID="ObjectsDetected" objects="{detections}" />

                <!-- Select object to pick -->
                <Action ID="SelectObject"
                        objects="{detections}"
                        selected="{target_object}" />

                <!-- Estimate pose -->
                <Action ID="EstimatePose"
                        object="{target_object}"
                        pose="{object_pose}" />

                <!-- Plan grasp -->
                <Action ID="PlanGrasp"
                        pose="{object_pose}"
                        grasp="{best_grasp}" />

                <!-- Execute pick with retry -->
                <RetryUntilSuccessful num_attempts="3">
                    <Action ID="ExecutePick" grasp="{best_grasp}" />
                </RetryUntilSuccessful>

                <!-- Execute place -->
                <Action ID="ExecutePlace" target="{place_pose}" />

                <!-- Update metrics -->
                <Action ID="LogSuccess" />

            </Sequence>
        </RepeatUntilFailure>

    </Sequence>
  </BehaviorTree>
</root>
```

1.9.2 7.2 Behavior Tree Node Implementations

```cpp
class DetectObjectsAction : public BT::SyncActionNode {
public:
    DetectObjectsAction(const std::string& name, const BT::NodeConfiguration& config)
        : BT::SyncActionNode(name, config) {

        // ROS2 node
        node_ = rclcpp::Node::make_shared("detect_objects_bt_node");

        // Service client
        client_ = node_->create_client<vision_interfaces::srv::DetectObjects>("/vision/detect_
    }

    static BT::PortsList providedPorts() {
        return { BT::OutputPort<std::vector<Detection>>("output") };
    }

    BT::NodeStatus tick() override {
        // Call vision service
        auto request = std::make_shared<vision_interfaces::srv::DetectObjects::Request>();

        if (!client_->wait_for_service(std::chrono::seconds(1))) {
            return BT::NodeStatus::FAILURE;
        }

        auto future = client_->async_send_request(request);

        if (rclcpp::spin_until_future_complete(node_, future) !=
            rclcpp::FutureReturnCode::SUCCESS) {
            return BT::NodeStatus::FAILURE;
        }

        auto response = future.get();

        if (response->detections.empty()) {
            return BT::NodeStatus::FAILURE;
        }

        // Set output port
        setOutput("output", response->detections);

        return BT::NodeStatus::SUCCESS;
    }

private:
    rclcpp::Node::SharedPtr node_;
```

```
    rclcpp::Client<vision_interfaces::srv::DetectObjects>::SharedPtr client_;
};
```

1.9.2.1 7.2.1 DetectObjects Action Node

1.9.3 7.3 State Machine (Alternative to BT)

Package: `task_orchestration` **File:** `src/state_machine.cpp`

```cpp
enum class SystemState {
    IDLE,
    SCANNING,
    DETECTING,
    PLANNING_GRASP,
    EXECUTING_PICK,
    EXECUTING_PLACE,
    ERROR,
    EMERGENCY_STOP
};

class TaskStateMachine : public rclcpp::Node {
public:
    TaskStateMachine() : Node("task_state_machine"), current_state_(SystemState::IDLE) {
        // State transition timer
        timer_ = create_wall_timer(
            std::chrono::milliseconds(100),
            std::bind(&TaskStateMachine::update, this));
    }

    void update() {
        switch (current_state_) {
            case SystemState::IDLE:
                handleIdle();
                break;
            case SystemState::SCANNING:
                handleScanning();
                break;
            case SystemState::DETECTING:
                handleDetecting();
                break;
            // ... (other states)
        }
    }

    void handleIdle() {
        if (trigger_received_) {
            transition(SystemState::SCANNING);
        }
```

```cpp
    }

    void handleScanning() {
        // Capture image
        auto future = vision_client_->async_send_request(/* request */);

        // Wait for response (non-blocking)
        if (future.wait_for(std::chrono::seconds(0)) == std::future_status::ready) {
            auto response = future.get();
            if (response->success) {
                transition(SystemState::DETECTING);
            } else {
                transition(SystemState::ERROR);
            }
        }
    }

    void transition(SystemState new_state) {
        RCLCPP_INFO(get_logger(), "State transition: %s -> %s",
                    stateToString(current_state_).c_str(),
                    stateToString(new_state).c_str());
        current_state_ = new_state;
        publishState();
    }

    void publishState() {
        std_msgs::msg::String msg;
        msg.data = stateToString(current_state_);
        state_pub_->publish(msg);
    }

    std::string stateToString(SystemState state) {
        static const std::map<SystemState, std::string> state_names = {
            {SystemState::IDLE, "IDLE"},
            {SystemState::SCANNING, "SCANNING"},
            {SystemState::DETECTING, "DETECTING"},
            // ...
        };
        return state_names.at(state);
    }

private:
    SystemState current_state_;
    rclcpp::TimerBase::SharedPtr timer_;
    rclcpp::Publisher<std_msgs::msg::String>::SharedPtr state_pub_;
    bool trigger_received_ = false;
};
```

1.10 8. Monitoring Subsystem

1.10.1 8.1 Metrics Collector

Package: monitoring **Node Name:** metrics_collector_node

```python
class MetricsCollector(Node):
    def __init__(self):
        super().__init__('metrics_collector')

        # Prometheus metrics
        self.pick_success_counter = Counter(
            'pick_success_total', 'Total successful picks')
        self.pick_failure_counter = Counter(
            'pick_failure_total', 'Total failed picks')
        self.cycle_time_histogram = Histogram(
            'cycle_time_seconds', 'Pick-place cycle time')
        self.grasp_quality_gauge = Gauge(
            'grasp_quality', 'Current grasp quality score')

        # InfluxDB client
        self.influx_client = InfluxDBClient(
            url=self.declare_parameter('influxdb_url', 'http://localhost:8086').value,
            token=self.declare_parameter('influxdb_token', '').value,
            org='robotics')
        self.write_api = self.influx_client.write_api(write_options=SYNCHRONOUS)

        # Subscribers
        self.create_subscription(
            TaskStatus, '/orchestrator/status', self.status_callback, 10)
        self.create_subscription(
            Grasp, '/grasp/selected', self.grasp_callback, 10)

        # Start Prometheus HTTP server
        start_http_server(8000)

        self.cycle_start_time = None

    def status_callback(self, msg: TaskStatus):
        """Track task status and cycle time."""
        if msg.state == "EXECUTING_PICK":
            self.cycle_start_time = self.get_clock().now()

        elif msg.state == "COMPLETED":
            if self.cycle_start_time:
                duration = (self.get_clock().now() - self.cycle_start_time).nanoseconds / 1e9
                self.cycle_time_histogram.observe(duration)
```

```python
                # Write to InfluxDB
                point = Point("cycle_time") \
                    .tag("robot_id", "robot_01") \
                    .field("duration", duration) \
                    .time(datetime.utcnow(), WritePrecision.NS)
                self.write_api.write(bucket="metrics", record=point)

            self.pick_success_counter.inc()

        elif msg.state == "ERROR":
            self.pick_failure_counter.inc()

    def grasp_callback(self, msg: Grasp):
        """Track grasp quality."""
        self.grasp_quality_gauge.set(msg.quality)
```

1.10.2 8.2 Database Schema

```sql
-- Database: robotics_db

-- Table: picks
CREATE TABLE picks (
    id SERIAL PRIMARY KEY,
    timestamp TIMESTAMP NOT NULL DEFAULT NOW(),
    robot_id VARCHAR(50) NOT NULL,
    object_id VARCHAR(100),
    object_class VARCHAR(50),
    pick_pose JSONB,  -- {x, y, z, qx, qy, qz, qw}
    place_pose JSONB,
    grasp_quality FLOAT,
    cycle_time FLOAT,  -- seconds
    success BOOLEAN,
    error_code VARCHAR(100),
    error_message TEXT
);

CREATE INDEX idx_picks_timestamp ON picks(timestamp DESC);
CREATE INDEX idx_picks_robot_id ON picks(robot_id);
CREATE INDEX idx_picks_success ON picks(success);

-- Table: system_health
CREATE TABLE system_health (
    id SERIAL PRIMARY KEY,
    timestamp TIMESTAMP NOT NULL DEFAULT NOW(),
    robot_id VARCHAR(50) NOT NULL,
    cpu_percent FLOAT,
```

```sql
    memory_percent FLOAT,
    disk_percent FLOAT,
    gpu_percent FLOAT,
    gpu_memory_mb FLOAT,
    temperature_celsius FLOAT
);

-- Table: calibrations
CREATE TABLE calibrations (
    id SERIAL PRIMARY KEY,
    timestamp TIMESTAMP NOT NULL DEFAULT NOW(),
    robot_id VARCHAR(50) NOT NULL,
    calibration_type VARCHAR(50),  -- 'hand_eye' | 'tcp' | 'workspace'
    parameters JSONB,
    reprojection_error FLOAT,
    notes TEXT
);
```

1.10.2.1 8.2.1 PostgreSQL Schema (Operational Data)

1.10.2.2 8.2.2 InfluxDB Schema (Time-Series Data)

```
# Measurement: cycle_time
# Tags: robot_id, object_class
# Fields: duration (float, seconds)

# Measurement: joint_states
# Tags: robot_id, joint_name
# Fields: position (float, radians), velocity (float, rad/s), effort (float, Nm)

# Measurement: vision_latency
# Tags: robot_id, stage (detect | pose_estimate)
# Fields: latency_ms (float)

# Measurement: force_torque
# Tags: robot_id
# Fields: fx, fy, fz, tx, ty, tz (float, Newtons/Nm)
```

1.11 9. Data Structures

1.11.1 9.1 Custom ROS2 Messages

Package: `robot_msgs`

1.11.1.1 9.1.1 ObjectDetection

```
# robot_msgs/msg/ObjectDetection.msg
std_msgs/Header header
```

```
string object_id
string class_name
float32 confidence
geometry_msgs/BoundingBox2D bbox
sensor_msgs/PointCloud2 pointcloud
```

1.11.1.2 9.1.2 GraspCandidate

```
# robot_msgs/msg/GraspCandidate.msg
geometry_msgs/PoseStamped pose
float32 quality           # [0, 1]
float32 gripper_width     # meters
float32 force             # Newtons
geometry_msgs/Vector3 approach_direction
float32[] grasp_matrix    # 6x6 wrench matrix (flattened)
```

1.11.1.3 9.1.3 TaskStatus

```
# robot_msgs/msg/TaskStatus.msg
std_msgs/Header header
string state              # IDLE | SCANNING | DETECTING | PLANNING | EXECUTING_PICK | EXECUTI
string message
float32 progress          # [0, 1]
int32 picks_completed
int32 picks_failed
float32 average_cycle_time
```

1.11.2 9.2 Configuration Data Structures

```yaml
# Workspace configuration (config/workspace.yaml)
workspace:
  pick_zone:
    min: [0.3, -0.4, 0.0]   # [x, y, z] in meters
    max: [0.7, 0.4, 0.3]
  place_zone:
    min: [-0.3, 0.3, 0.0]
    max: [0.1, 0.7, 0.3]
  safety_zone:
    type: "cylinder"
    center: [0.0, 0.0, 0.0]
    radius: 1.5  # meters
    height: 2.0

objects:
  - class_name: "red_cube"
    shape: "box"
    dimensions: [0.05, 0.05, 0.05]  # [width, depth, height]
    weight: 0.2  # kg
```

```yaml
  - class_name: "blue_cylinder"
    shape: "cylinder"
    dimensions: [0.04, 0.08]  # [radius, height]
    weight: 0.15
```

1.12 10. Algorithms

1.12.1 10.1 Inverse Kinematics (Analytical for UR5e)

```cpp
class UR5eAnalyticalIK {
public:
    /**
     * Compute all IK solutions for a given target pose.
     *
     * @param target_pose Target end-effector pose (4x4 homogeneous matrix)
     * @return Vector of joint solutions (each 6 DOF)
     */
    std::vector<std::array<double, 6>> solve(const Eigen::Matrix4d& target_pose) {
        std::vector<std::array<double, 6>> solutions;

        // DH parameters for UR5e
        const double d1 = 0.1625;
        const double a2 = -0.425;
        const double a3 = -0.3922;
        const double d4 = 0.1333;
        const double d5 = 0.0997;
        const double d6 = 0.0996;

        // Extract target position and orientation
        Eigen::Vector3d p_target = target_pose.block<3, 1>(0, 3);
        Eigen::Matrix3d R_target = target_pose.block<3, 3>(0, 0);

        // Solve for theta1 (2 solutions)
        double p05_x = p_target.x() - d6 * R_target(0, 2);
        double p05_y = p_target.y() - d6 * R_target(1, 2);

        for (int i = 0; i < 2; ++i) {
            double theta1 = std::atan2(p05_y, p05_x) + std::acos(d4 / std::hypot(p05_x, p05_y)
            if (i == 1) theta1 += M_PI;

            // Solve for theta5 (2 solutions per theta1)
            for (int j = 0; j < 2; ++j) {
                double theta5 = std::acos((p_target.x() * std::sin(theta1) -
                                           p_target.y() * std::cos(theta1) - d4) / d6);
                if (j == 1) theta5 = -theta5;
```

```cpp
                // Solve for theta6
                double theta6 = /* ... */;

                // Solve for theta2, theta3, theta4 (closed-form)
                // ... (full derivation omitted)

                std::array<double, 6> solution = {theta1, theta2, theta3, theta4, theta5, theta

                // Validate solution (joint limits, workspace)
                if (isValid(solution)) {
                    solutions.push_back(solution);
                }
            }
        }

        return solutions;  // Up to 8 solutions
    }

private:
    bool isValid(const std::array<double, 6>& joints) {
        // Check joint limits
        const std::array<std::pair<double, double>, 6> limits = {
            {-2*M_PI, 2*M_PI},  // shoulder_pan
            {-2*M_PI, 2*M_PI},  // shoulder_lift
            {-M_PI, M_PI},      // elbow
            {-2*M_PI, 2*M_PI},  // wrist_1
            {-2*M_PI, 2*M_PI},  // wrist_2
            {-2*M_PI, 2*M_PI}   // wrist_3
        };

        for (size_t i = 0; i < 6; ++i) {
            if (joints[i] < limits[i].first || joints[i] > limits[i].second) {
                return false;
            }
        }
        return true;
    }
};
```

1.12.2 10.2 Trajectory Smoothing (Cubic Spline)

```python
def smooth_trajectory(waypoints: List[np.ndarray], dt: float = 0.01) -> np.ndarray:
    """
    Generate smooth trajectory using cubic splines.

    Args:
        waypoints: List of joint configurations (each 6 DOF)
```

```
        dt: Time step (seconds)

    Returns:
        Smoothed trajectory as (N, 6) array
    """
    n_joints = len(waypoints[0])
    n_waypoints = len(waypoints)

    # Create time stamps for waypoints (assume unit spacing)
    t_waypoints = np.arange(n_waypoints)

    # Interpolate each joint independently
    smooth_traj = []
    for joint_idx in range(n_joints):
        joint_waypoints = [wp[joint_idx] for wp in waypoints]

        # Cubic spline interpolation
        cs = CubicSpline(t_waypoints, joint_waypoints, bc_type='clamped')

        # Evaluate at fine time steps
        t_fine = np.arange(0, n_waypoints - 1, dt)
        joint_traj = cs(t_fine)

        smooth_traj.append(joint_traj)

    # Transpose to (N, 6) shape
    return np.array(smooth_traj).T
```

1.12.3 10.3 Force Closure Check

```
def check_force_closure(contact_points: np.ndarray,
                        normals: np.ndarray,
                        friction_coeff: float = 0.5) -> bool:
    """
    Check if grasp achieves force closure.

    Args:
        contact_points: (N, 3) array of contact point positions
        normals: (N, 3) array of contact normals
        friction_coeff: Friction coefficient

    Returns:
        True if force closure is satisfied
    """
    n_contacts = len(contact_points)

    if n_contacts < 4:
```

```python
        return False  # Need at least 4 contacts for 3D force closure

    # Build grasp matrix G (6 x 3N)
    # Each contact contributes 3 columns (normal force + 2 tangential forces)
    G = []
    for i in range(n_contacts):
        r = contact_points[i]  # Position vector
        n = normals[i]         # Normal vector

        # Tangent vectors (perpendicular to normal)
        t1 = np.array([-n[1], n[0], 0])
        t1 /= np.linalg.norm(t1)
        t2 = np.cross(n, t1)

        # Force columns (normal + friction cone)
        for direction in [n, t1 * friction_coeff, t2 * friction_coeff]:
            force_col = direction
            torque_col = np.cross(r, direction)
            wrench_col = np.hstack([force_col, torque_col])
            G.append(wrench_col)

    G = np.array(G).T  # Shape: (6, 3N)

    # Check if G has full rank (rank = 6)
    rank = np.linalg.matrix_rank(G)

    return rank == 6
```

1.13 11. ROS2 Node Specifications

1.13.1 11.1 Vision Pipeline Nodes

Node	Executable	Topics (Pub)	Topics (Sub)	Services	Parameters
realsense_camera	realsense_driver	/camera/color/image_raw/camera/depth/image_rect_raw/camera/color/camer			width, height, fps, serial_number
image_processor	image_preprocessor	/vision/preprocessed/image	/camera/color/image_raw		denoise, enhance_contrast
yolo_detector	object_detector	/vision/detections/vision/debug/detections	/vision/preprocessed/image		model_path, confidence_threshold, use_tensorrt
pose_estimator	pose_estimator	/vision/object_poses	/vision/detections/camera/depth/image_rect_raw/camera/		method (pca | icp | pvnet)

1.13.2 11.2 Grasp Planning Nodes

Node	Executable	Services (Server)	Topics (Sub)	Parameters
grasp_synthesizer	grasp_planner	/compute_grasps	/vision/object_poses	num_candidates, gripper_max_width
grasp_evaluator	grasp_ranker	/rank_grasps	-	quality_metric (ferrari_canny | volume)

1.13.3 11.3 Motion Planning Nodes

Node	Executable	Actions (Server)	Topics (Pub)	Parameters
pick_place_server	motion_planner	/pick_place	/planned_path (visualization)	planner (rrtconnect | rrtstar)
collision_objects	scene_manager	-	/planning_scene	workspace_bounds

---

1.14 12. Database Schema

(Covered in Section 8.2)

1.15 13. API Specifications

1.15.1 13.1 REST API (FastAPI Backend)

Base URL: `http://localhost:8080/api/v1`

1.15.1.1 13.1.1 Endpoints

Method	Endpoint	Description	Request Body	Response
GET	/system/status	Get system status	-	{"state": "IDLE", "uptime": 3600, "picks_today": 120}
POST	/system/start	Start system	-	{"success": true}
POST	/system/stop	Stop system	-	{"success": true}
POST	/system/estop	Emergency stop	-	{"success": true}

Method	Endpoint	Description	Request Body	Response
GET	/picks	Get pick history	Query: ?limit=100&offset=0	[{id, timestamp, success, ...}]
GET	/picks/{id}	Get pick details	-	{id, object_class, cycle_time, ...}
GET	/metrics	Get metrics summary	-	{success_rate, avg_cycle_time, uptime}
POST	/calibration/hand-eye	Start calibration	{"num_poses": 10}	{calibration_id: "abc123"}
GET	/config	Get configuration	-	{workspace, objects, ...}
PUT	/config	Update configuration	{workspace: {...}}	{"success": true}

```python
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel
from typing import List, Optional

app = FastAPI(title="Robot Control API", version="1.0.0")

class SystemStatus(BaseModel):
    state: str
    uptime: float
    picks_today: int
    success_rate: float

class PickRecord(BaseModel):
    id: int
    timestamp: str
    object_class: str
    cycle_time: float
    success: bool
    error_message: Optional[str] = None

@app.get("/api/v1/system/status", response_model=SystemStatus)
async def get_system_status():
    """Get current system status."""
    # Query database or ROS2 topic
    status = query_system_status()
    return SystemStatus(**status)
```

```python
@app.post("/api/v1/system/start")
async def start_system():
    """Start the robotic system."""
    # Publish ROS2 message to start system
    success = publish_ros_command("START")
    if not success:
        raise HTTPException(status_code=500, detail="Failed to start system")
    return {"success": True}

@app.get("/api/v1/picks", response_model=List[PickRecord])
async def get_picks(limit: int = 100, offset: int = 0):
    """Get pick history with pagination."""
    picks = query_database(
        "SELECT * FROM picks ORDER BY timestamp DESC LIMIT %s OFFSET %s",
        (limit, offset))
    return [PickRecord(**p) for p in picks]
```

1.15.1.2 13.1.2 FastAPI Implementation

1.15.2 13.2 gRPC API (Performance-Critical Operations)

Proto File: `robot_interfaces/proto/robot_control.proto`

```protobuf
syntax = "proto3";

package robot_control;

service RobotController {
    rpc GetJointStates(Empty) returns (JointStates);
    rpc MoveJoints(JointPositions) returns (MoveResponse);
    rpc ExecuteGrasp(GraspPose) returns (GraspResponse);
    rpc StreamMetrics(Empty) returns (stream Metrics);
}

message JointStates {
    repeated double positions = 1;  // 6 DOF
    repeated double velocities = 2;
    repeated double efforts = 3;
    int64 timestamp = 4;
}

message JointPositions {
    repeated double positions = 1;
    double max_velocity = 2;
    double max_acceleration = 3;
}

message MoveResponse {
```

```
    bool success = 1;
    string message = 2;
    double execution_time = 3;
}

message Metrics {
    double cycle_time = 1;
    double success_rate = 2;
    int32 picks_today = 3;
}
```

1.16 14. State Machines

(Covered in Section 7.3)

1.17 15. Error Handling

1.17.1 15.1 Error Hierarchy

```
RobotError
   VisionError
      CameraConnectionError
      ObjectNotDetectedError
      PoseEstimationError
   PlanningError
      IKFailureError
      CollisionError
      PathPlanningTimeoutError
   ExecutionError
      TrajectoryExecutionError
      GraspFailureError
      ControllerError
   HardwareError
      RobotCommunicationError
      GripperError
      SensorError
   SafetyError
       CollisionDetectedError
       ForceExceededError
       EmergencyStopError
```

1.17.2 15.2 Error Recovery Strategies

```
class ErrorRecovery:
    """Centralized error recovery manager."""
```

```python
    def handle_error(self, error: RobotError) -> bool:
        """
        Handle error with appropriate recovery strategy.

        Returns:
            True if recovery successful, False otherwise
        """
        if isinstance(error, GraspFailureError):
            return self.retry_grasp_with_adjustment()

        elif isinstance(error, ObjectNotDetectedError):
            return self.adjust_lighting_and_retry()

        elif isinstance(error, IKFailureError):
            return self.replan_with_different_approach()

        elif isinstance(error, SafetyError):
            self.emergency_stop()
            return False

        else:
            self.log_error(error)
            return False

    def retry_grasp_with_adjustment(self) -> bool:
        """Retry grasp with increased force or different approach."""
        for attempt in range(3):
            # Increase force by 10%
            force = self.current_grasp.force * 1.1

            # Retry
            success = self.execute_grasp(force=force)
            if success:
                return True

        return False
```

1.17.3 15.3 Logging and Alerting

```python
import logging
from logging.handlers import RotatingFileHandler

# Configure logging
logger = logging.getLogger('robot_system')
logger.setLevel(logging.DEBUG)
```

```python
# File handler (rotating, max 10 MB, keep 5 backups)
fh = RotatingFileHandler(
    '/var/log/robot/system.log', maxBytes=10*1024*1024, backupCount=5)
fh.setLevel(logging.DEBUG)

# Console handler
ch = logging.StreamHandler()
ch.setLevel(logging.INFO)

# Formatter
formatter = logging.Formatter(
    '%(asctime)s - %(name)s - %(levelname)s - %(message)s')
fh.setFormatter(formatter)
ch.setFormatter(formatter)

logger.addHandler(fh)
logger.addHandler(ch)

# Usage
logger.info("System started")
logger.warning("Object detection confidence low: 0.45")
logger.error("Grasp execution failed", exc_info=True)
logger.critical("Safety limit exceeded: Force = 200N")
```

1.18 16. Performance Optimization

1.18.1 16.1 Latency Budget

Component	Target Latency	Optimization
Image Capture	<10 ms	Hardware trigger, pre-allocated buffers
Object Detection	<50 ms	TensorRT FP16, batch inference
Pose Estimation	<30 ms	GPU-accelerated PCA, downsampled point cloud
Grasp Planning	<200 ms	Cached grasp database, parallel candidate evaluation
Motion Planning	<300 ms	RRT* with early termination, warm-start IK
Trajectory Execution	<1000 ms	Time-optimal parameterization
Total Cycle Time	<2000 ms	**Achieved: 1620 ms**

1.18.2 16.2 CPU/GPU Optimization

```cpp
// CUDA kernel for parallel grasp quality evaluation
__global__ void evaluateGraspsKernel(
```

```
    const Grasp* grasps, float* qualities, int n_grasps) {

    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx >= n_grasps) return;

    // Compute grasp quality metric in parallel
    qualities[idx] = computeFerrariCanny(grasps[idx]);
}

void evaluateGraspsParallel(
    const std::vector<Grasp>& grasps, std::vector<float>& qualities) {

    int n = grasps.size();

    // Allocate device memory
    Grasp* d_grasps;
    float* d_qualities;
    cudaMalloc(&d_grasps, n * sizeof(Grasp));
    cudaMalloc(&d_qualities, n * sizeof(float));

    // Copy to device
    cudaMemcpy(d_grasps, grasps.data(), n * sizeof(Grasp), cudaMemcpyHostToDevice);

    // Launch kernel
    int blockSize = 256;
    int numBlocks = (n + blockSize - 1) / blockSize;
    evaluateGraspsKernel<<<numBlocks, blockSize>>>(d_grasps, d_qualities, n);

    // Copy back
    qualities.resize(n);
    cudaMemcpy(qualities.data(), d_qualities, n * sizeof(float), cudaMemcpyDeviceToHost);

    // Free
    cudaFree(d_grasps);
    cudaFree(d_qualities);
}
```

1.19 17. Testing Hooks

1.19.1 17.1 Mock Interfaces

```
// Virtual interface for robot driver (allows mocking)
class IRobotDriver {
public:
    virtual ~IRobotDriver() = default;
    virtual void moveJoints(const std::vector<double>& positions) = 0;
```

```cpp
    virtual std::vector<double> getJointPositions() = 0;
};

// Real implementation
class URRobotDriver : public IRobotDriver {
    void moveJoints(const std::vector<double>& positions) override {
        // Send to real robot via TCP/IP
    }
};

// Mock for testing
class MockRobotDriver : public IRobotDriver {
    void moveJoints(const std::vector<double>& positions) override {
        last_commanded_positions_ = positions;
    }

    std::vector<double> last_commanded_positions_;
};

// Usage in tests
TEST(MotionPlannerTest, TestJointMove) {
    auto mock_driver = std::make_shared<MockRobotDriver>();
    MotionPlanner planner(mock_driver);

    std::vector<double> target = {0, -1.57, 0, -1.57, 0, 0};
    planner.moveToJointPositions(target);

    EXPECT_EQ(mock_driver->last_commanded_positions_, target);
}
```

1.19.2 17.2 Dependency Injection

```python
class VisionPipeline:
    def __init__(self, detector=None, pose_estimator=None):
        self.detector = detector or YOLODetector()
        self.pose_estimator = pose_estimator or PCABasedPoseEstimator()

    def process_image(self, image):
        detections = self.detector.detect(image)
        poses = self.pose_estimator.estimate(detections)
        return poses

# Test with mock detector
class MockDetector:
    def detect(self, image):
        return [Detection(class_name="cube", bbox=[100, 100, 200, 200])]
```

```python
def test_vision_pipeline():
    pipeline = VisionPipeline(detector=MockDetector())
    poses = pipeline.process_image(dummy_image)
    assert len(poses) == 1
```

1.20 18. Deployment Configuration

1.20.1 18.1 Docker Compose

```yaml
# docker-compose.yml
version: '3.8'

services:
  vision:
    image: robotics/vision:latest
    runtime: nvidia  # GPU support
    devices:
      - /dev/video0:/dev/video0  # Camera
    environment:
      - ROS_DOMAIN_ID=42
      - CUDA_VISIBLE_DEVICES=0
    volumes:
      - ./models:/models:ro
      - ./config:/config:ro
    networks:
      - robot_net
    restart: unless-stopped

  motion_planning:
    image: robotics/motion_planning:latest
    environment:
      - ROS_DOMAIN_ID=42
    volumes:
      - ./config:/config:ro
    networks:
      - robot_net
    restart: unless-stopped

  orchestrator:
    image: robotics/orchestrator:latest
    environment:
      - ROS_DOMAIN_ID=42
    networks:
      - robot_net
    restart: unless-stopped
```

```yaml
  database:
    image: postgres:15
    environment:
      - POSTGRES_DB=robotics_db
      - POSTGRES_USER=robot
      - POSTGRES_PASSWORD=${DB_PASSWORD}
    volumes:
      - db_data:/var/lib/postgresql/data
    networks:
      - robot_net

  grafana:
    image: grafana/grafana:10.0.0
    ports:
      - "3000:3000"
    volumes:
      - grafana_data:/var/lib/grafana
      - ./dashboards:/etc/grafana/provisioning/dashboards:ro
    networks:
      - robot_net

networks:
  robot_net:
    driver: bridge

volumes:
  db_data:
  grafana_data:
```

1.20.2 18.2 Kubernetes Deployment

```yaml
# k8s/deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: vision-pipeline
spec:
  replicas: 1
  selector:
    matchLabels:
      app: vision
  template:
    metadata:
      labels:
        app: vision
    spec:
      containers:
```

```yaml
      - name: vision
        image: robotics/vision:latest
        resources:
          limits:
            nvidia.com/gpu: 1
        env:
        - name: ROS_DOMAIN_ID
          value: "42"
        volumeMounts:
        - name: models
          mountPath: /models
          readOnly: true
      volumes:
      - name: models
        persistentVolumeClaim:
          claimName: models-pvc
```

1.21 19. Summary

This Low-Level Design provides comprehensive technical specifications for implementing all subsystems:

1.21.1 19.1 Key Deliverables

Vision Subsystem: Camera driver, object detector (YOLOv8 + TensorRT), pose estimator (PCA) **Grasp Planning:** Antipodal grasp synthesis, force closure validation, collision checking **Motion Planning:** MoveIt2 integration, pick-place action server, trajectory optimization **Control:** ros2_control hardware interface, 1 kHz control loop, safety limits **Orchestration:** Behavior tree implementation, state machine, error recovery **Monitoring:** Prometheus/InfluxDB metrics, PostgreSQL operational DB, REST/gRPC APIs

1.21.2 19.2 Performance Targets

Metric	Target	Implementation
Cycle Time	<2 sec	Optimized pipeline, TensorRT inference, parallel processing
Vision Latency	<50 ms	GPU acceleration, FP16 precision
Motion Planning	<300 ms	RRT* with warm-start, simplified collision meshes
Control Frequency	1 kHz	RT-Linux, FIFO scheduling, pre-allocated memory
Success Rate	>99%	Robust grasp planning, adaptive error recovery

1.21.3 19.3 Next Steps

1. **Implementation:** Use this LLD as blueprint for coding
2. **Unit Testing:** Implement test cases with 80%+ coverage
3. **Integration Testing:** Validate subsystem interfaces
4. **Performance Profiling:** Measure and optimize latency bottlenecks
5. **Documentation:** Generate API docs from code (Doxygen/Sphinx)

Document Status: v1.0 Complete **Next Document:** C4 Model Diagrams (architectural visualizations) **Dependencies:** High-Level Design (08), Technical Stack (05), Problem Statement (04)
