Robot Control System

Architecture Documentation

Version 1.0

Modern React-Based Robotics Framework

June 15, 2025

Contents

1 Introduction

The Robot Control System is a modern, web-based robotics control framework built with React and Three.js. It provides a comprehensive solution for robot visualization, control, and manipulation through a clean, context-based architecture that separates concerns while maintaining high performance for real-time operations.

1.1 System Overview

The system enables users to:

- Discover and load URDF robot models dynamically
- Control robot joints in real-time with visual feedback
- Attach and configure TCP (Tool Center Point) tools
- Record and playback robot trajectories
- Solve inverse kinematics with multiple solver algorithms
- Manage multiple robots in a shared workspace
- Persist workspace configurations across sessions

1.2 Key Technologies

- **React 18**: Modern UI framework with hooks and context
- **Three.js**: 3D graphics and visualization
- **URDF**: Universal Robot Description Format support
- **EventBus**: High-frequency update system
- **Context API**: State management architecture
- **Custom Hooks**: Clean API abstractions

1.3 Design Philosophy

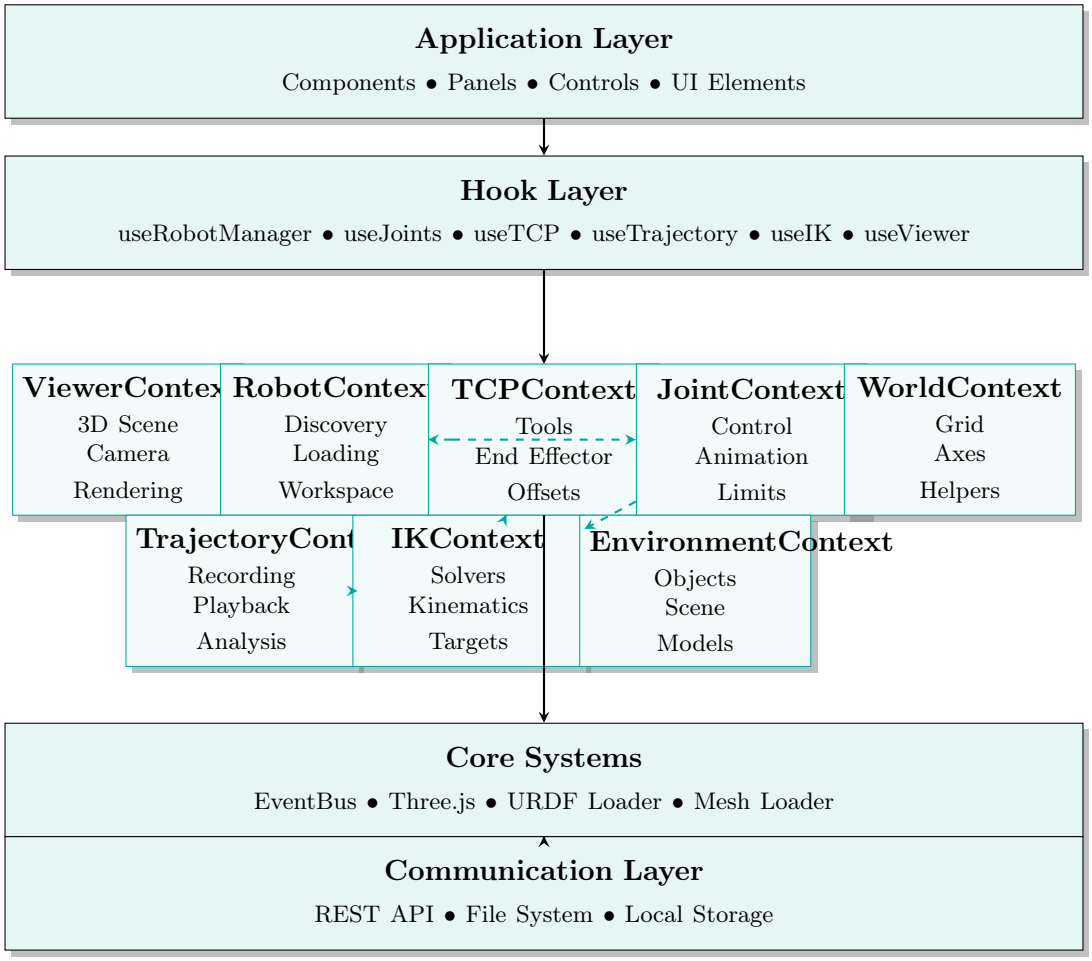
The architecture follows these core principles:

1. **Separation of Concerns**: Each context manages a single domain
2. **Performance Optimization**: EventBus for real-time, React state for UI
3. **Developer Experience**: Intuitive hooks and clear patterns
4. **Extensibility**: Easy to add new features and contexts
5. **Type Safety Ready**: Structured for TypeScript migration
6. **Professional Grade**: Production-ready architecture

2 Architecture Overview

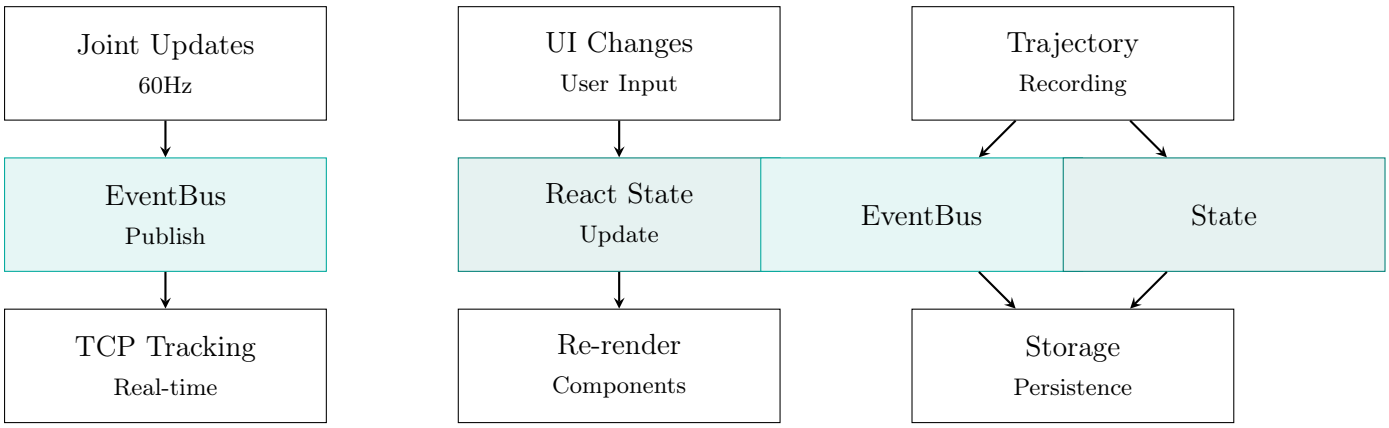
2.1 System Architecture

The Robot Control System implements a sophisticated layered architecture:



2.2 Data Flow Architecture

The system implements a hybrid data flow pattern optimized for different update frequencies:



3 Core Architecture Components

3.1 Context System

The application uses React Context API as the primary state management solution, with each context managing a specific domain:

3.1.1 ViewerContext

Manages the 3D visualization layer:

```
const ViewerContext = createContext(null);

export const ViewerProvider = ({ children }) => {
  const [isViewerReady, setIsViewerReady] = useState(false);
  const [viewerConfig, setViewerConfig] = useState({
    backgroundColor: '#f5f5f5',
    enableShadows: true,
    ambientColor: '#8ea0a8',
    upAxis: '+Z',
    highlightColor: '#ff0000'
  });

  const sceneSetupRef = useRef(null);
  const dragControlsRef = useRef(null);

  // Initialize 3D scene
  const initializeViewer = useCallback((container, config = {}) => {
    if (!container || sceneSetupRef.current) return;

    const sceneSetup = new SceneSetup({
      container,
      backgroundColor: config.backgroundColor,
      enableShadows: config.enableShadows
    });

    sceneSetupRef.current = sceneSetup;
    setIsViewerReady(true);

    EventBus.emit('viewer:initialized', { sceneSetup });
    return sceneSetup;
  }, []);

  // ... additional methods
};
```

Listing 1: ViewerContext Structure

3.1.2 RobotContext

The unified foundation managing all robot-related operations:

Responsibility	Description
Discovery	Fetches available robots from server API
Workspace	Manages user's selected robots with persistence
Loading	Handles URDF loading and 3D scene integration
Joint Control	Delegates joint control to loaded robots
Active Management	Tracks currently selected robot
Tool Discovery	Discovers available TCP tools

Table 1: RobotContext Responsibilities

3.2 Hook Layer

The hook layer provides clean APIs for components to interact with contexts:

```
export const useRobotManager = () => {
  const context = useRobotContext();

  return {
    // State
    availableRobots: context.availableRobots,
    workspaceRobots: context.workspaceRobots,
    activeRobotId: context.activeRobotId,
    loadedRobots: context.loadedRobots,

    // Operations
    discoverRobots: context.discoverRobots,
    addRobotToWorkspace: context.addRobotToWorkspace,
    loadRobot: context.loadRobot,
    setActiveRobotId: context.setActiveRobotId,

    // Joint Control
    setJointValue: context.setJointValue,
    getJointValues: context.getJointValues,

    // Computed Properties
    hasRobots: context.hasRobots,
    isLoading: context.isLoading
  };
};
```

Listing 2: useRobotManager Hook

3.3 EventBus System

For high-frequency updates, the system uses a custom EventBus:

```
class EventBusClass {
  constructor() {
    this.events = {};
  }

  on(event, handler) {
    if (!this.events[event]) {
      this.events[event] = [];
```

```
    }
    this.events[event].push(handler);

    // Return unsubscribe function
    return () => {
      this.events[event] = this.events[event]
        .filter(h => h !== handler);
    };
  }

  emit(event, data) {
    if (!this.events[event]) return;
    this.events[event].forEach(handler => handler(data));
  }
}

const EventBus = new EventBusClass();
export default EventBus;
```

Listing 3: EventBus Implementation

4 Context Implementations

4.1 TCP Context

Manages Tool Center Point operations and end effector calculations:

```
class OptimizedTCPManager {
  calculateRobotEndEffector(robotId) {
    const robot = this.findRobot(robotId);
    if (!robot) return { position: {x:0,y:0,z:0} };

    // Find end effector link
    const endEffectorNames = [
      'tool0', 'ee_link', 'end_effector',
      'gripper_link', 'link_6', 'link_7'
    ];

    let endEffectorLink = null;
    for (const name of endEffectorNames) {
      if (robot.links[name]) {
        endEffectorLink = robot.links[name];
        break;
      }
    }

    // Get world position
    const worldPos = new THREE.Vector3();
    endEffectorLink.getWorldPosition(worldPos);

    return {
      position: { x: worldPos.x, y: worldPos.y, z: worldPos.z }
    };
  }
```

```
28
29    getFinalEndEffectorPosition(robotId) {
30      const robotEndEffector = this.calculateRobotEndEffector(robotId);
31      const toolData = this.attachedTools.get(robotId);
32
33      if (!toolData) return robotEndEffector.position;
34
35      const tipOffset = this.calculateToolTipOffset(toolData.
            toolContainer);
36
37      return {
38        x: robotEndEffector.position.x + tipOffset.x,
39        y: robotEndEffector.position.y + tipOffset.y,
40        z: robotEndEffector.position.z + tipOffset.z
41      };
42    }
43  }
```

Listing 4: TCP Context Core Methods

4.2 Joint Context

Handles joint control and animation with sophisticated interpolation:

```
1  const animateToJointValues = useCallback(async (robotId, targetValues
       , options = {}) => {
2    const {
3      duration = 1000,
4      tolerance = 0.001,
5      easing = 'exponential'
6    } = options;
7
8    return new Promise((resolve) => {
9      const currentValues = getRobotJointValues(robotId);
10     const startTime = Date.now();
11
12     const easingFunctions = {
13       linear: t => t,
14       exponential: t => 1 - Math.exp(-5 * t),
15       smoothstep: t => t * t * (3 - 2 * t)
16     };
17
18     const animate = () => {
19       const elapsed = Date.now() - startTime;
20       const progress = Math.min(elapsed / duration, 1);
21       const smoothProgress = easingFunctions[easing](progress);
22
23       // Interpolate values
24       const interpolatedValues = {};
25       Object.keys(targetValues).forEach(jointName => {
26         const start = currentValues[jointName] || 0;
27         const end = targetValues[jointName];
28         interpolatedValues[jointName] = start + (end - start) *
               smoothProgress;
29       });
30
```

```
      // Apply to robot
      setRobotJointValues_Internal(robotId, interpolatedValues);

      if (progress >= 1) {
        resolve({ success: true });
      } else {
        requestAnimationFrame(animate);
      }
    };

    requestAnimationFrame(animate);
  });
}, []);
```

Listing 5: Joint Animation System

4.3 Trajectory Context

Records and plays back robot movements using optimized data collection:

```
const startRecording = useCallback((trajectoryName, robotId,
    dataCallback, interval = 100) => {
  const recordingState = {
    trajectoryName,
    robotId,
    startTime: Date.now(),
    frames: [],
    endEffectorPath: [],
    isRecording: true,
    dataCallback
  };

  setRecordingStates(prev => new Map(prev).set(robotId,
      recordingState));

  const intervalId = setInterval(() => {
    if (recordingState.isRecording) {
      const currentTime = Date.now() - recordingState.startTime;
      const frameData = dataCallback();

      if (frameData) {
        recordingState.frames.push({
          timestamp: currentTime,
          jointValues: frameData.jointValues || {}
        });

        if (frameData.endEffectorPosition) {
          recordingState.endEffectorPath.push({
            timestamp: currentTime,
            position: frameData.endEffectorPosition,
            orientation: frameData.endEffectorOrientation
          });
        }
      }
    }
  }, interval);
```

```
35
36    recordingIntervalsRef.current.set(robotId, intervalId);
37    return true;
38  }, []);
```

Listing 6: Trajectory Recording

4.4 IK Context

Manages inverse kinematics with dynamic solver loading:

```
const loadSolver = useCallback(async (solverName) => {
  if (solversRef.current[solverName]) return true;

  try {
    // Fetch solver code dynamically
    const response = await fetch(`/IKSolvers/${solverName}.jsx`);
    const solverCode = await response.text();

    // Remove imports and create solver
    const modifiedCode = solverCode
      .replace(/import\s+.*?from\s+['"].*?['"];?/g, '')
      .replace(/export\s+default\s+/, 'return ');

    const createSolver = new Function('THREE', modifiedCode);
    const SolverClass = createSolver(THREE);

    const solverInstance = new SolverClass(SolverClass.defaultConfig
       || {});
    solversRef.current[solverName] = solverInstance;

    return true;
  } catch (error) {
    console.error(`Failed to load solver ${solverName}:`, error);
    return false;
  }
}, []);
```

Listing 7: IK Solver Integration

5 Performance Optimization Strategies

5.1 Hybrid State Management

The system uses different state management approaches based on update frequency:

Data Type	Management	Use Case	Frequency
Joint Values	EventBus	Real-time control	60Hz
TCP Position	EventBus	End effector tracking	30-60Hz
UI State	React State	Component updates	On change
Robot List	React State	Discovery results	Once
Trajectories	Hybrid	Recording/playback	Variable

Table 2: State Management Strategy

5.2 Optimization Techniques

5.2.1 Registry Pattern

Local registries cache frequently accessed data:

```
const robotRegistryRef = useRef(new Map());

const findRobotWithFallbacks = useCallback((robotId) => {
  // Check local registry first (fastest)
  if (robotRegistryRef.current.has(robotId)) {
    return robotRegistryRef.current.get(robotId);
  }

  // Try robot manager methods
  if (robotManagerRef.current?.getRobot) {
    const robot = robotManagerRef.current.getRobot(robotId);
    if (robot) {
      // Cache for future use
      robotRegistryRef.current.set(robotId, robot);
      return robot;
    }
  }

  // Scene traversal as last resort
  // ... fallback logic
}, []);
```

Listing 8: Robot Registry Pattern

5.2.2 Memoization

Expensive computations are memoized:

```
// Memoize robot filtering
const industrialRobots = useMemo(() => {
  return availableRobots.filter(robot =>
    robot.type === 'industrial' && robot.dof >= 6
  );
}, [availableRobots]);

// Memoize callbacks
const getRobotById = useCallback((robotId) => {
  return availableRobots.find(robot => robot.id === robotId);
}, [availableRobots]);
```

Listing 9: Memoization Examples

6 Hook Design Patterns

6.1 Main Hook Pattern

Each context provides a main hook that exposes all functionality:

```
export const useRobotManager = () => {
  const context = useRobotContext();

  return {
    // Direct state exposure
    ...context,

    // Computed properties
    hasRobots: context.robots.size > 0,

    // Convenience methods
    getRobotById: useCallback((id) => {
      return context.availableRobots.find(r => r.id === id);
    }, [context.availableRobots])
  };
};
```

Listing 10: Main Hook Pattern

6.2 Specialized Hook Pattern

Focused hooks for specific use cases:

```
export const useRobotWorkspace = () => {
  const manager = useRobotManager();

  return {
    robots: manager.workspaceRobots,
    addRobot: manager.addRobotToWorkspace,
    removeRobot: manager.removeRobotFromWorkspace,
    hasRobot: manager.hasWorkspaceRobot
  };
};

export const useRobotSelection = () => {
  const manager = useRobotManager();

  return {
    activeId: manager.activeRobotId,
    setActive: manager.setActiveRobotId,
    isActive: manager.isRobotActive
  };
};
```

Listing 11: Specialized Hooks

6.3 Robot-Specific Hook Pattern

Many features need robot-specific data:

```
export const useJoints = (robotId = null) => {
  const context = useJointContext();
  const { activeId } = useRobotSelection();

  // Use provided robotId or fall back to active
  const targetRobotId = robotId || activeId;

  // Get robot-specific data
  const jointInfo = targetRobotId ?
    context.getJointInfo(targetRobotId) : [];
  const jointValues = targetRobotId ?
    context.getJointValues(targetRobotId) : {};

  // Robot-specific methods
  const setJointValue = useCallback((name, value) => {
    return context.setJointValue(targetRobotId, name, value);
  }, [context, targetRobotId]);

  return {
    robotId: targetRobotId,
    jointInfo,
    jointValues,
    setJointValue,
    hasJoints: jointInfo.length > 0
  };
};
```

Listing 12: Robot-Specific Pattern

7 Event System

7.1 Event Naming Convention

Events follow a consistent naming pattern:

Pattern	Example	Use Case
`domain:action`	`'robot:loaded'`	Robot loaded
`domain:state-change`	`'tcp:tool-attached'`	Tool attached
`domain:error`	`'ik:solver-failed'`	Error occurred

Table 3: Event Naming Convention

7.2 Event Flow Examples

7.2.1 Joint Update Flow

```
// 1. User moves joint slider
const handleJointChange = (jointName, value) => {
```

```
  setJointValue(jointName, value);
};

// 2. Joint Context updates robot
const setJointValue = (robotId, jointName, value) => {
  const success = robot.setJointValue(jointName, value);

  if (success) {
    // 3. Emit event for other systems
    EventBus.emit('robot:joint-changed', {
      robotId,
      jointName,
      value,
      allValues: getJointValues(robotId)
    });
  }
};

// 4. TCP Context listens and updates
useEffect(() => {
  const handleJointChange = (data) => {
    if (data.robotId === targetRobotId) {
      recalculateEndEffector(data.robotId);
    }
  };

  const unsubscribe = EventBus.on('robot:joint-changed',
     handleJointChange);
  return () => unsubscribe();
}, [targetRobotId]);
```

Listing 13: Joint Update Event Flow

8 UI Component Integration

8.1 Component Structure

Components use hooks to access system functionality:

```
const RobotControl = () => {
  const {
    workspaceRobots,
    activeRobotId,
    loadRobot,
    setActiveRobotId
  } = useRobotManager();

  const {
    jointValues,
    setJointValue,
    hasJoints
  } = useJoints();

  const handleRobotSelect = async (robot) => {
```

```
    if (!robot.loaded) {
      await loadRobot(robot.id, robot.urdfPath);
    }
    setActiveRobotId(robot.id);
  };

  return (
    <div className="robot-control">
      <RobotSelector
        robots={workspaceRobots}
        activeId={activeRobotId}
        onSelect={handleRobotSelect}
      />

      {hasJoints && (
        <JointControl
          joints={jointValues}
          onChange={setJointValue}
        />
      )}
    </div>
  );
};
```

Listing 14: Robot Control Component

8.2 Provider Setup

The application wraps components with context providers:

```
const App = () => {
  return (
    <ViewerProvider>
      <RobotProvider>
        <EnvironmentProvider>
          <TCPProvider>
            <JointProvider>
              <TrajectoryProvider>
                <IKProvider>
                  <WorldProvider>
                    <AppContent />
                  </WorldProvider>
                </IKProvider>
              </TrajectoryProvider>
            </JointProvider>
          </TCPProvider>
        </EnvironmentProvider>
      </RobotProvider>
    </ViewerProvider>
  );
};
```

Listing 15: Application Provider Chain

9 API Integration

9.1 Robot Discovery API

The system communicates with a backend server for robot discovery:

```
const discoverRobots = useCallback(async () => {
  try {
    setIsLoading(true);

    const response = await fetch('/robots/list');
    const result = await response.json();

    if (result.success) {
      const data = result.categories || [];
      setCategories(data);

      const allRobots = [];
      data.forEach(category => {
        category.robots.forEach(robot => {
          allRobots.push({
            ...robot,
            category: category.id,
            categoryName: category.name
          });
        });
      });

      setAvailableRobots(allRobots);
    }
  } catch (err) {
    setError('Failed to discover robots');
  } finally {
    setIsLoading(false);
  }
}, []);
```

Listing 16: Robot Discovery

9.2 File System Integration

URDF and mesh files are loaded dynamically:

```
const loadRobot = useCallback(async (robotName, urdfPath, options =
    {}) => {
  const loader = new URDFLoader();

  // Configure loader
  loader.packages = urdfPath.substring(0, urdfPath.lastIndexOf('/'));
  loader.loadMeshCb = (path, manager, done) => {
    MeshLoader.load(path, manager, (obj, err) => {
      if (err) return done(null, err);

      // Configure materials
      obj.traverse(child => {
```

```
        if (child instanceof THREE.Mesh) {
          child.castShadow = true;
          child.receiveShadow = true;
        }
      });

      done(obj);
    });
  };

  // Load URDF
  const robot = await new Promise((resolve, reject) => {
    loader.load(urdfPath, resolve, null, reject);
  });

  return robot;
}, []);
```

Listing 17: URDF Loading

10 Best Practices

10.1 Context Design Guidelines

1. **Single Responsibility**: Each context manages one domain
2. **Clear Dependencies**: Document context relationships
3. **Avoid Circular Dependencies**: Use EventBus for cross-context communication
4. **Provide Defaults**: Always have sensible default values
5. **Error Boundaries**: Wrap providers with error boundaries
6. **Cleanup**: Always clean up listeners and intervals

10.2 Hook Design Guidelines

1. **Consistent Naming**: Use `use` prefix
2. **Return Objects**: For clarity and extensibility
3. **Memoize**: Use `useCallback` and `useMemo`
4. **Handle Loading**: Expose loading and error states
5. **Document**: Use JSDoc comments
6. **Test**: Write unit tests for hooks

10.3 Performance Guidelines

1. **EventBus for Real-time**: Use for >10Hz updates
2. **State for UI**: Use React state for UI updates
3. **Batch Updates**: Group related state changes
4. **Lazy Loading**: Load contexts only when needed
5. **Profile**: Use React DevTools Profiler
6. **Optimize Renders**: Use React.memo when appropriate

11 Future Roadmap

11.1 Planned Features

1. **WebSocket Integration**: Real-time robot control
2. **Multi-Robot Coordination**: Collision avoidance
3. **Cloud Persistence**: Save workspaces to cloud
4. **Collaboration**: Multi-user support
5. **Plugin System**: Dynamic feature loading
6. **TypeScript Migration**: Full type safety

11.2 Architecture Evolution

The architecture is designed to scale:

- **Microservices**: Split backend into services
- **GraphQL**: Replace REST with GraphQL
- **WebAssembly**: Performance-critical calculations
- **AR/VR Support**: Immersive control interfaces
- **AI Integration**: Smart trajectory planning
- **ROS Bridge**: Connect to ROS systems

12 Conclusion

The Robot Control System represents a modern approach to web-based robotics control. By leveraging React's context system, Three.js for visualization, and a carefully designed architecture that separates concerns while maintaining performance, the system provides a solid foundation for building sophisticated robotics applications.

The hybrid state management approach ensures real-time performance for critical operations while maintaining the benefits of React's declarative programming model for UI updates. The

extensive hook system provides clean APIs that make the complex underlying systems accessible to developers.

As the system evolves, the modular architecture ensures that new features can be added without disrupting existing functionality, making it an ideal platform for both research and production robotics applications.