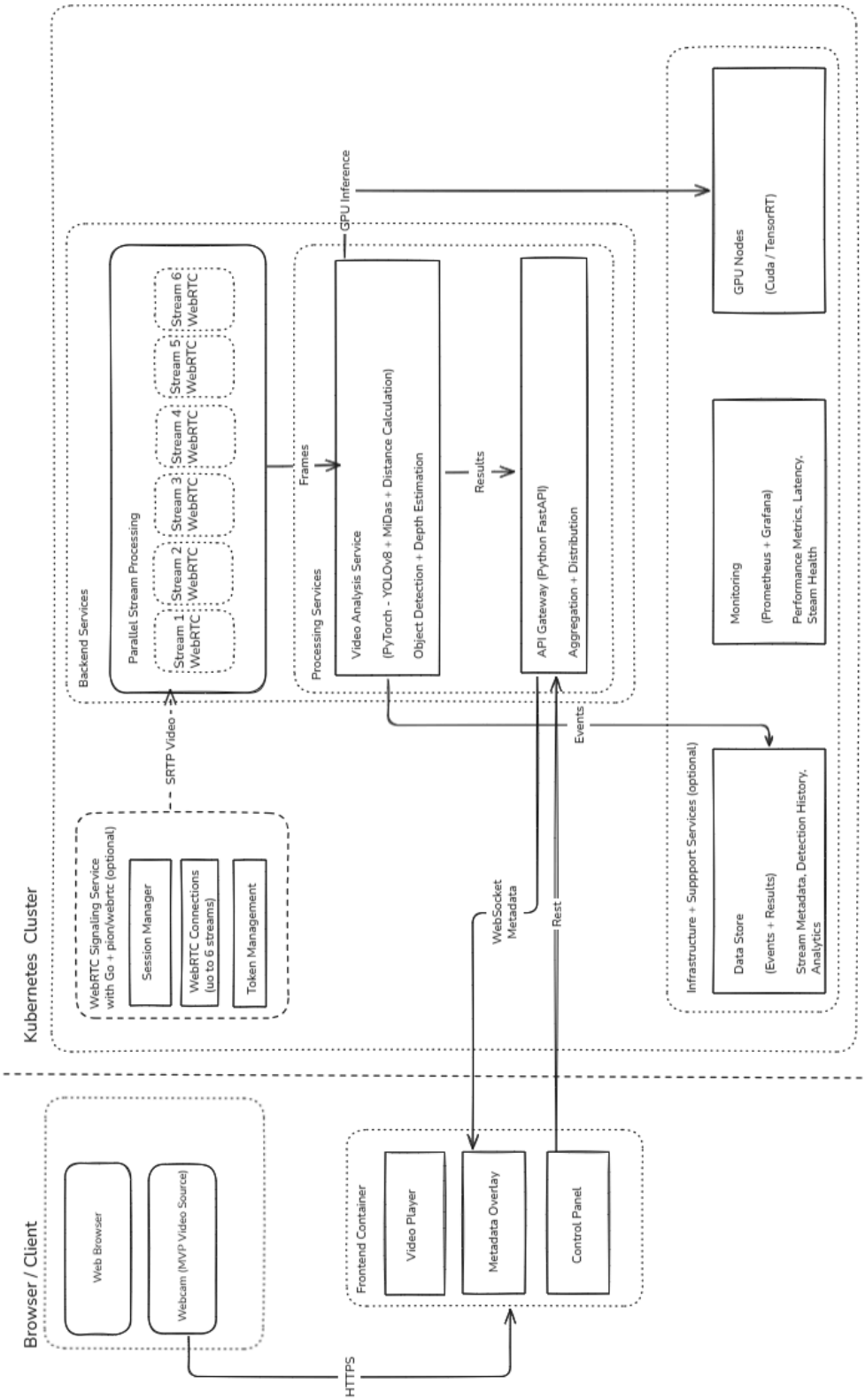
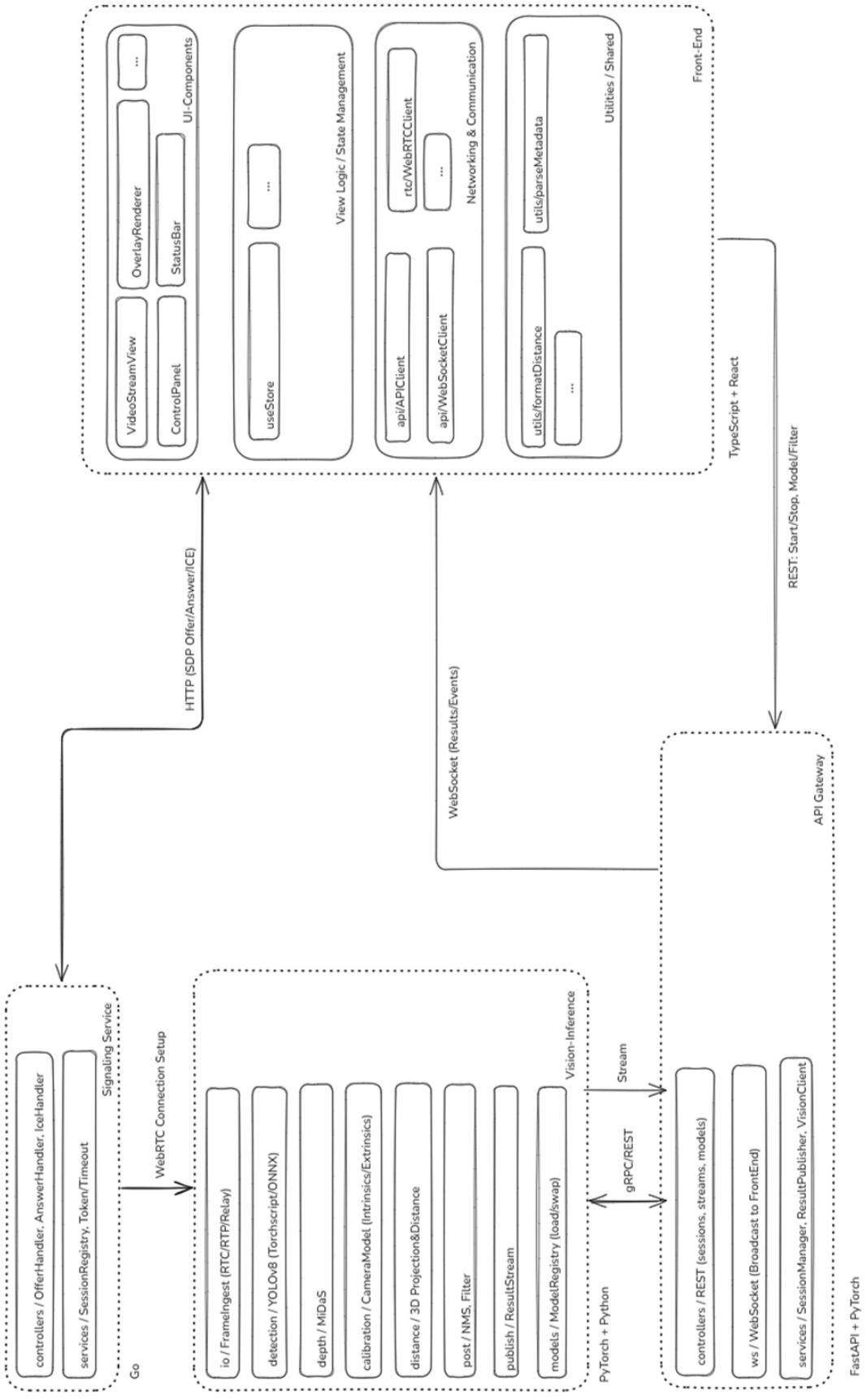
Page 1: Overview diagram of runtime components



Page 2: Overview diagram of code components



Page 3: Summary of the underlying technology stack

The system is designed for real-time object detection and depth estimation in video streams from mobile robots. It analyzes camera images and provides operators with information about detected objects and their distances through a web interface.

React and TypeScript form the basis of the frontend. WebRTC displays live video streams in the browser with low latency. A video element shows the stream directly, while a Canvas shows object detection results as overlays. Each detected object appears with a bounding box and distance information.

The Python-backend is implemented with FastAPI, which handles WebRTC video streaming through the aiortc library. This component manages the WebRTC signaling process, establishes peer connections with browsers, and streams video from the robot's camera. REST endpoints provide basic control and status queries.

Python and PyTorch power the core image analysis. YOLOv8 through the ultralytics package detects objects in each video frame. MiDaS estimates depth from the monocular camera image. Both models work together to calculate approximate 3D positions and distances for each detected object. Results are sent to the frontend for visualization.

Docker containers package all application components. This ensures consistent deployment across different environments. For production, Kubernetes orchestrates the containers, enabling scalable deployment and management. During development, services run locally to simplify testing and debugging.

This stack focuses on technologies that are well-documented and widely used. Python unifies most components, which simplifies development and troubleshooting. WebRTC keeps latency low enough for monitoring and teleoperation without requiring separate streaming infrastructure.

Page 4: Textual explanation of the diagrams and choices

The runtime diagram shows how components interact during operation. Browser clients establish direct WebRTC connections with the backend service running in a Kubernetes cluster. This keeps latency minimal. The backend coordinates multiple concurrent streams and distributes video frames to AI processing workers. Each stream maintains its own processing pipeline, allowing up to six parallel streams without interference.

The signaling service appears with dashed boundaries because it remains unclear. WebRTC connection setup currently runs within the Python backend through aiortc. A separate Go-based signaling service using Pion WebRTC can be deployed later if performance optimization becomes necessary. The modular design supports both configurations without major changes.

Monitoring infrastructure with Prometheus and Grafana tracks system health and ensures latency requirements are met. These components are optional during development but recommended for production deployments.

The code structure emphasizes modularity and testability. Frontend components separate concerns clearly: video reception, canvas rendering for overlays, and state management. The backend organizes around FastAPI for REST endpoints and aiortc for WebRTC connections. AI processing isolates model loading from inference execution, allowing models to be loaded once and reused across all streams.

Communication between components uses different protocols for different purposes. WebRTC carries video data with minimal latency. REST endpoints handle configuration and control commands. WebSockets can transmit detection results and metadata in real time, though this remains planned rather than fully implemented.

Python unifies most backend components because the AI ecosystem centers around Python libraries. PyTorch, YOLOv8, and MiDaS all provide mature Python APIs. The aiortc library handles WebRTC without requiring a separate language stack. This consistency simplifies development and troubleshooting compared to mixing Python and Go services.

React and TypeScript provide the frontend foundation. TypeScript adds type safety to JavaScript, catching errors during development rather than runtime. WebRTC APIs work reliably in modern browsers, and the Canvas element renders detection overlays efficiently.

FastAPI supports async operations naturally. Its automatic API documentation and WebSocket support reduce development time. Docker containers package services for consistent deployment across environments, while later Kubernetes should orchestrate containers in production.