Chapter Outline

Architecture and Development Workflows

Chapter 2 Experimental Platform Configuration

2.1 Key Tools

2.2 Installation and First Simulation

2.3 Architecture and Workflows

2.4 API and SDK Interfaces

2.5 Examples and Experimental Routes

2.6 Exercises and Discussion

4 Units in This Section

2.3.1 Overall Architecture and Core Components
Computational roles, feedback data, and observation sources

2.3.2 Model and Algorithm Development
Motion models, control software, and high-level applications

2.3.3 Simulation Modes and Transfer to Real Systems
Software, target hardware, real vehicles, and swarm transfer

2.3.4 Comparative Testing and Results Analysis
Parameter comparisons, fault propagation, and interpretation

Textbook: Sec. 2.2.1–2.2.2 (architecture and components); Sec. 2.3.1–2.3.3 (low-level control, fault testing, perception, and swarms)

2.3 Architecture and Development Workflows: Introduction

From a Running System to Your Development Task

- **Runtime operation:** Trace the motion target through controller output, model motion, and sensor feedback to see how the tools jointly close the loop.
- **Development scope:** Change inertia in the motion model, control laws in the autopilot, or recognition methods in a high-level application. Each produces a different runtime artifact.
- **Validation:** Model tests reveal motion behavior, software tests check program interaction, and target-hardware tests reveal computation and communication timing. The research question determines the test conditions.

Identify the Component → **Build the Artifact** → **Connect to the Loop** → **Compare Results**

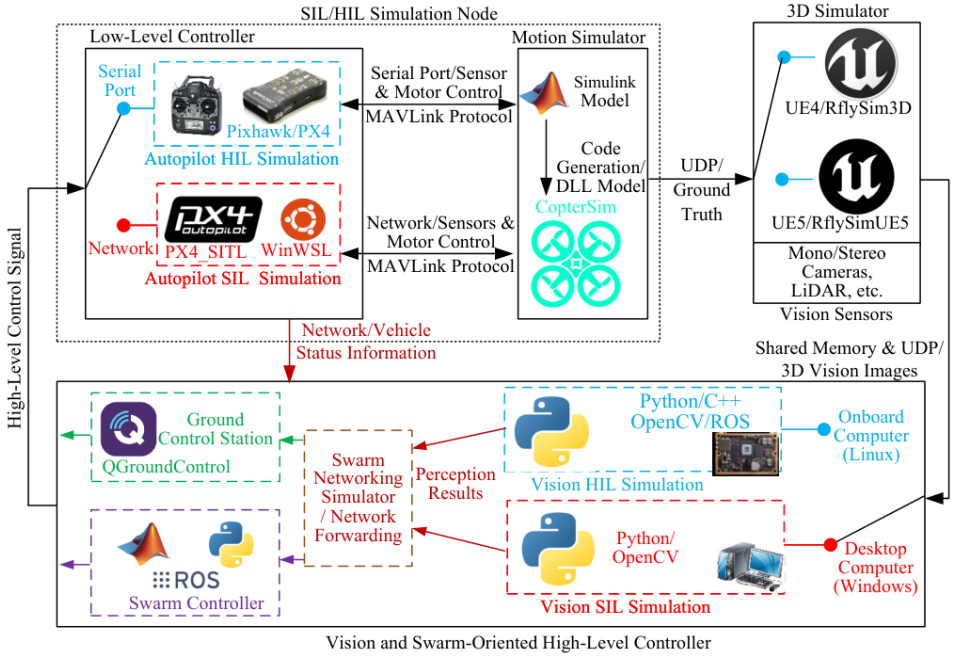
Example: Change inertia → Build a DLL (dynamic-link library) → Load in CopterSim → Compare angular rates.

Identify what you will develop, then define its output, integration point, and validation method.

2.3 Architecture and Development Workflows

2.3.1 Overall Architecture and Core Components: Overall Framework

Runtime Architecture: Four Types of Computation Close the Loop



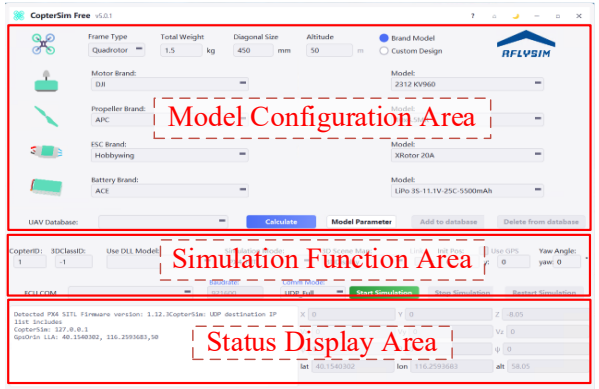
- **Vehicle motion simulator:** CopterSim computes pose and virtual sensor measurements from actuator commands.
- **Low-level controller:** The autopilot fuses measurements, then uses targets and estimated states to compute actuator commands.
- **3D simulator:** Updates the scene from pose and generates images, depth, and other perception data.
- **High-level controller:** Recognizes targets, plans motion, and sends new targets to the autopilot.

Fig. 2.9 Overall Framework of the RflySim Toolchain
Software-in-the-loop (SITL; textbook SIL); hardware-in-the-loop (HITL; textbook HIL)

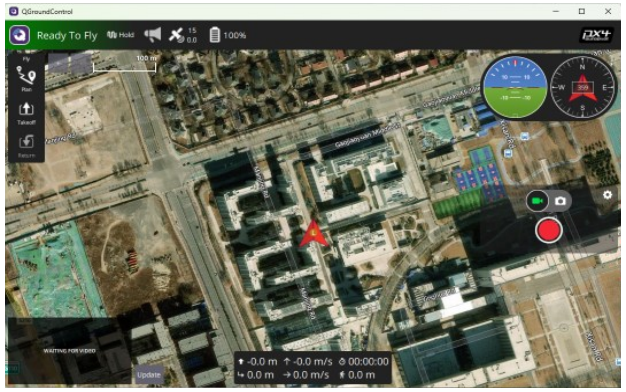
Models compute motion and measurements; controllers use feedback to drive the next motion step.

2.3.1 Overall Architecture and Core Components: Component Roles

Software Windows Reflect Distinct Computations



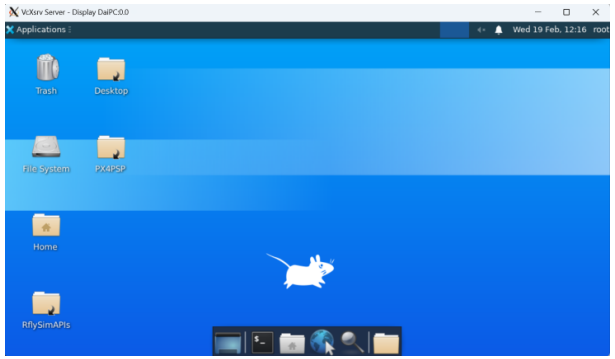
(a) CopterSim



(b) QGroundControl



(c) RflySim3D



(d) WinWSL/WslGUI

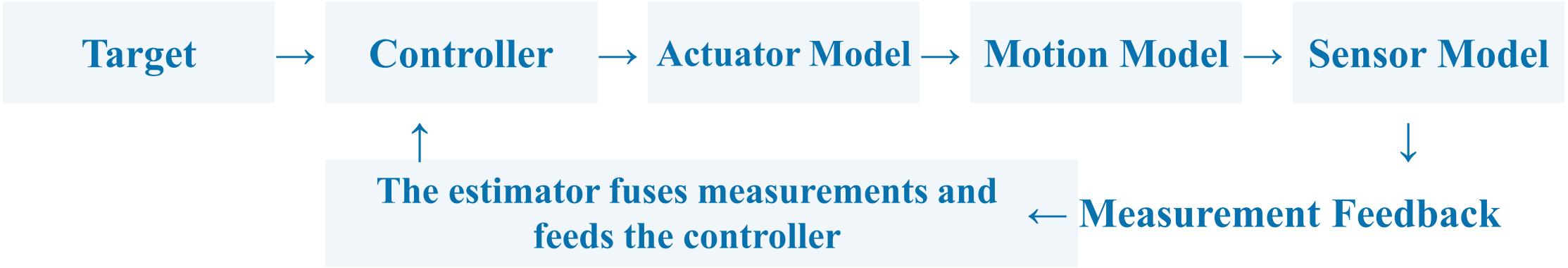
Fig. 2.11 Core Software Interface Previews

- **CopterSim:** Loads the dynamics model, shows its state, and links autopilot, 3D simulator, and high-level programs as a communication hub.
- **QGroundControl:** Receives autopilot telemetry, displays estimated states, and configures parameters and mission routes.
- **RflySim3D:** Maps pose to scene motion and outputs images and other observations according to sensor settings.
- **WinWSL:** Provides a Linux environment for PX4 SITL, ROS, and OpenCV programs, and can compile PX4 firmware.

2.3.1 Overall Architecture and Core Components: Low-Level Control

Low-Level Loop: Commands Become New Measurements

Closing the loop: Measurements return to the controller and help determine its next actuator output.



- **Actuation to motion:** Motor or control-surface models turn commands into forces and torques. The motion model then updates velocity, position, and attitude.
- **Measurement to control:** Sensor models generate measurements, which the estimator fuses into states. The controller uses the target–estimate error to adjust the next output.

Actuators change motion; measurements return to the estimator, and the controller adjusts its output.

2.3.1 Overall Architecture and Core Components: Low-Level Control

Low-Level Loop: Iteration and Error Convergence

After step: error < 5 mm in ~900 cycles (3.6 s); hover thrust 13.7 N

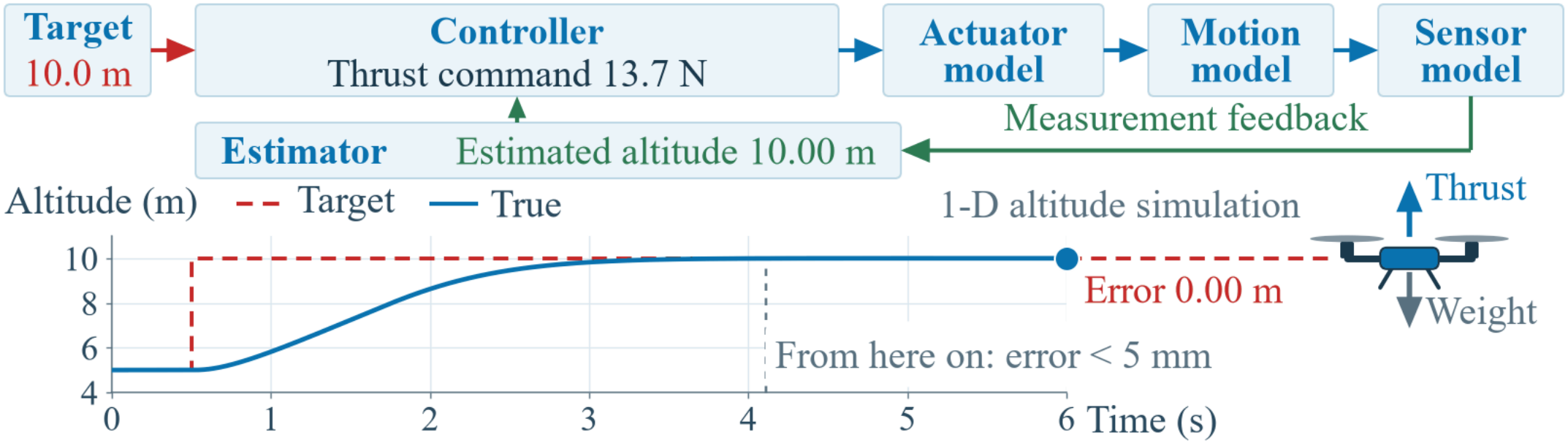


Figure: Step-by-Step Convergence of the Low-Level Loop (Simplified 1D Altitude Simulation)

Measurements feed the next computation step; altitude error gradually converges in this example.

2.3.1 Overall Architecture and Core Components: Perception and Decision-Making

High-Level Loop: Scene Observations Produce New Motion Targets

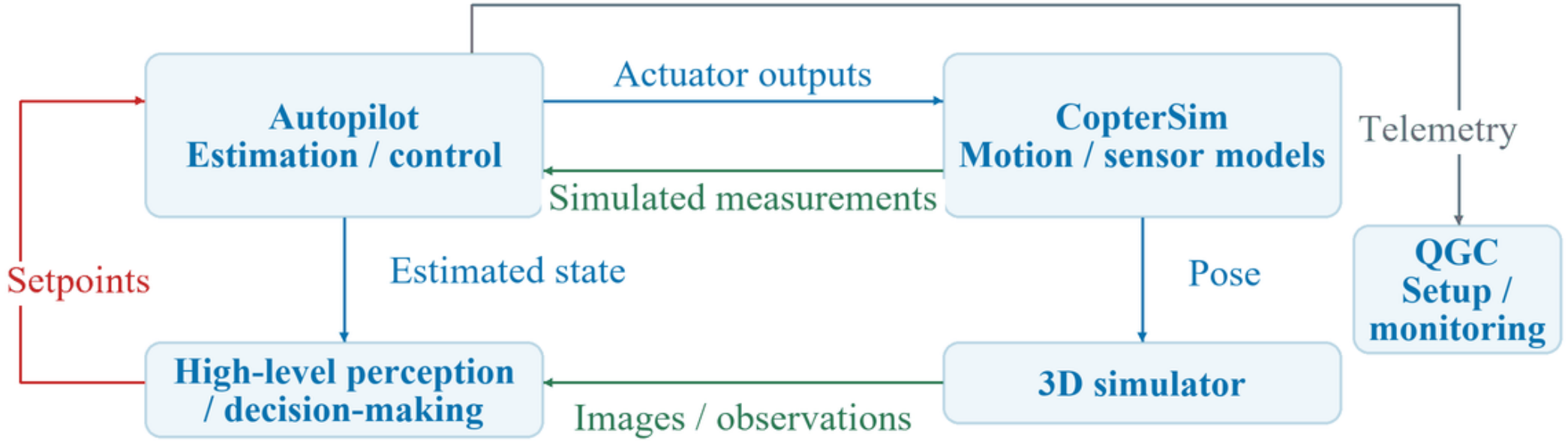


Figure: Coupled Low-Level Control and High-Level Perception–Decision Loops

Motion feedback: Measurements enter the autopilot, while model pose drives the scene.

Task feedback: Algorithms recognize targets, and planning results return to the autopilot.

Low-level control maintains motion; high-level perception and decisions act via the setpoint interface.

2.3.1 Overall Architecture and Core Components: High-Level Controller

High-Level Control for Vehicles and Swarms

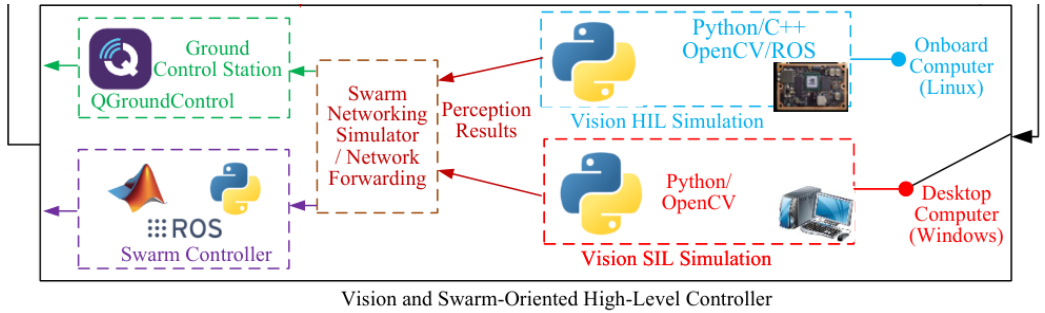


Fig. 2.9 Overall Framework of the RflySim Toolchain (Detail)

- **Perception/decision layer:** Uses vehicle images and point clouds for detection, tracking, localization, mapping, obstacle avoidance, and task decisions.
- **Networking/communication layer:** The swarm network simulator relays vehicle states and perception data.
- **Swarm control layer:** The ground station and swarm decision controller coordinate vehicles using centralized or distributed control (next page).

Inputs: Images and point clouds from the 3D simulator, and state estimates from the autopilot.
Outputs: Commands such as arming, takeoff, desired velocity, or trajectory.

High-level control turns observations into motion targets; networked vehicles share data to coordinate.

2.3.1 Overall Architecture and Core Components: Swarm Control

Centralized and Distributed Swarm Control Architectures

Criterion	Centralized	Distributed
Decision Location	A ground station collects all states and jointly plans trajectories and tasks	Each node runs its own decision logic using local and neighbor information
Benefits and Costs	Supports unified global optimization; depends on communication and is vulnerable to interference	Less central dependence; flexible scaling; coordination and communication constraints remain
Suitable Tasks	Smaller tasks with high accuracy requirements	Large swarms or communication-limited environments

Choose by scale, communication, and real-time needs; the two architectures can also be combined.

Distributed control reduces central dependence but demands more coordination over intervehicle links.

2.3.1 Overall Architecture and Core Components: Model Responsibilities

Dynamics, 3D Appearance, and Sensors Have Distinct Roles

Component	Behavior It Determines	Example Change
Dynamics Model	Actuator commands → Forces, motion, and simulated measurements	Changing inertia changes the response through the model equations
3D Model and Scene	Geometry, materials, environment, and visible appearance	Changing appearance does not automatically change dynamics parameters
Sensor Configuration	Mounting pose, field of view, resolution, and data output	Rotating the camera changes its observation area

Example: CopterID identifies the instance; ClassID selects its 3D appearance. Both must match the dynamics model's vehicle.

2.3.1 Overall Architecture and Core Components: Data Sources

Ground Truth, Observations, and Estimates Can Differ

Data	Source	Main Use
Simulation Ground Truth	States in the motion or scene model	Drives visualization and provides a reference for error analysis
Sensor Observations	Sensor models and their sampling process	Simulate what an algorithm reads under specified conditions
State Estimates	Autopilot or high-level estimator	Fuse observations for control and task programs

Example: Model position → Noisy positioning measurement → Fused position estimate.

Measurements come from model states; estimates fuse measurements, so one motion can yield different values.

Chapter Outline

Architecture and Development Workflows

Chapter 2 Experimental Platform Configuration

2.1 Key Tools

2.2 Installation and First Simulation

2.3 Architecture and Workflows

2.4 API and SDK Interfaces

2.5 Examples and Experimental Routes

2.6 Exercises and Discussion

4 Units in This Section

2.3.1 Overall Architecture and Core Components
Computational roles, feedback data, and observation sources

2.3.2 Model and Algorithm Development
Motion models, control software, and high-level applications

2.3.3 Simulation Modes and Transfer to Real Systems
Software, target hardware, real vehicles, and swarm transfer

2.3.4 Comparative Testing and Results Analysis
Parameter comparisons, fault propagation, and interpretation

Textbook: Sec. 2.2.1–2.2.2 (architecture and components); Sec. 2.3.1–2.3.3 (low-level control, fault testing, perception, and swarms)

2.3.2 Model and Algorithm Development: Tools and Artifacts

Development Artifacts Have Different Build and Runtime Locations

Development and Build		Deliverable		Runtime Location
Motion Modeling Simulink model and parameters	→	**Model DLL** Dynamic-link library and parameters	→	**CopterSim** Motion and simulated measurements
Control Design Simulink or C/C++ modules	→	**Control Software/Firmware** Generate, integrate, and build	→	**Autopilot** Estimation, control, and scheduling
Perception and Decisions Python／C++／ROS	→	**Application + Configuration** Program, weights, interface settings	→	**Desktop/Onboard Computer** Process observations; output task targets

Runtime example: CopterSim loads the model DLL directly; after model changes, rebuild and load the new file.

2.3 Architecture and Development Workflows

2.3.2 Model and Algorithm Development: Development Overview

Development Connects Prototypes, Runtime Artifacts, and Real Hardware

- **Model and control (1–3):** In Simulink, 1 build the motion model, 2 design the controller, and 3 close the simulation loop. Tune using tracking error and overshoot.
- **Code generation (4):** Build controller code into PX4 software; generate a motion-model DLL for CopterSim.
- **Hardware simulation (5):** Download to the autopilot; exchange sensor data and control signals with CopterSim over USB.
- **Real-vehicle experiments (6):** After installation, test on the bench, indoors, then in the field; tune against the physical response.

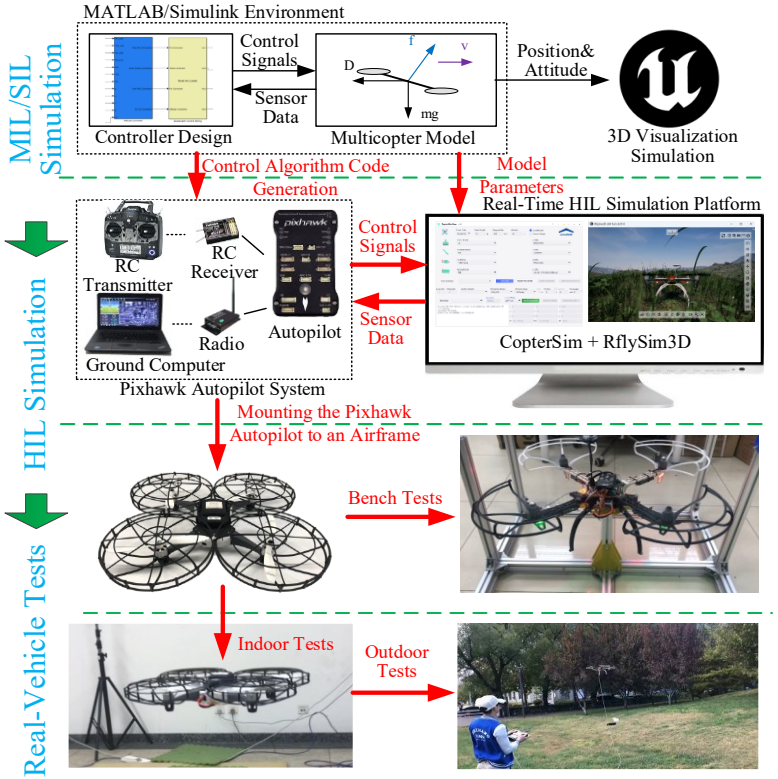


Fig. 2.17 Low-Level Control Algorithm Development and Experimental Workflow
Model-in-the-loop (MIL)

Validate with models; integrate the corresponding artifacts for software, hardware, and real-vehicle tests.

2.3 Architecture and Development Workflows

2.3.2 Model and Algorithm Development: Development Responsibilities

Motion, Scenes, Control, and Program Integration Are Developed Separately

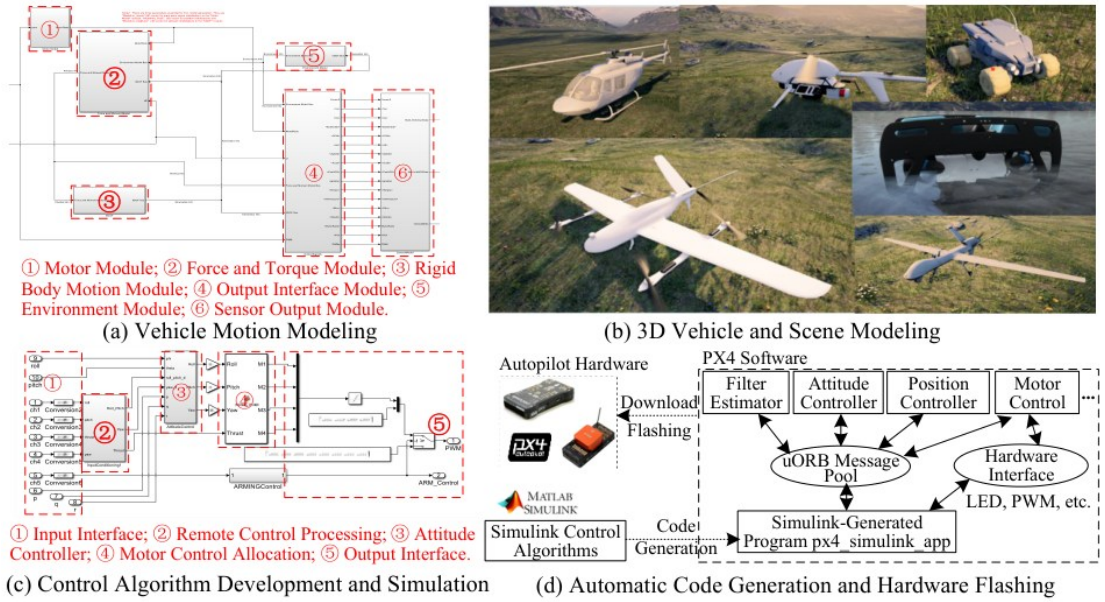


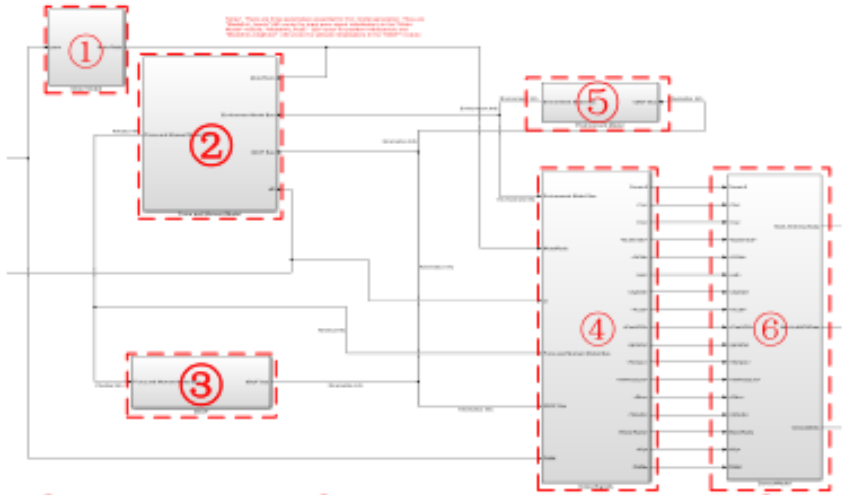
Fig. 2.18 Key Processes in Low-Level Control Algorithm Development

- **Motion modeling (a):** Model actuators, forces, and motion equations to output vehicle states and simulated measurements.
- **Scene modeling (b):** Set geometry, materials, and the environment for 3D visualization and sensor observations.
- **Control design (c):** Compute control effort from targets and estimated states, then allocate it among actuators.
- **Program integration (d):** Connect algorithm inputs/outputs, generate code, and build it into target software or firmware.

Four development tasks create motion, observation, and control capabilities; interfaces form the system.

2.3.2 Model and Algorithm Development: Motion Model

Motion Models Turn Actuator Inputs into States and Observations



① Motor Module; ② Force and Torque Module; ③ Rigid Body Motion Module; ④ Output Interface Module; ⑤ Environment Module; ⑥ Sensor Output Module.

Fig. 2.18(a) Motion Modeling

- **Actuation and forces:** Motor model ① computes speed; ② combines thrust, gravity, and other forces and torques.
- **Motion and environment:** ③ updates pose and velocity; ⑤ supplies altitude-dependent air density and other environmental values.
- **Output and execution:** ④ packages states; ⑥ generates sensor measurements. CopterSim calls the compiled DLL periodically.

Examples: e2-UavModeling (Dynamic Modeling); e2_DLL-Load (DLL Model Import)

Model equations determine the response; the compiled DLL brings the same model into CopterSim.

2.3.2 Model and Algorithm Development: Scenes and Perception

Scene Observations Provide Controlled Inputs for Perception Algorithms



Fig. 2.18(b) 3D Vehicle and Scene Modeling

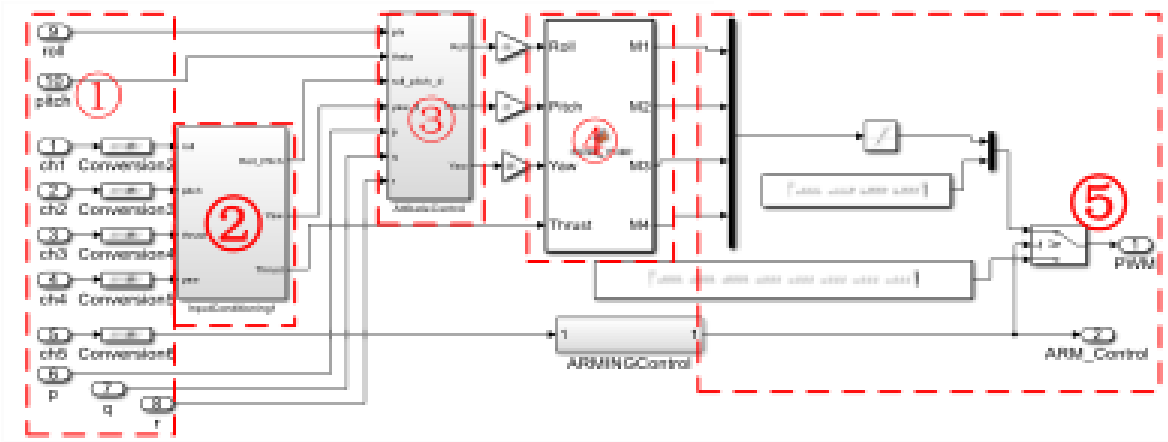
- **Controlled environment:** Geometry, materials, lighting, and occlusion determine sensor observations.
- **Multimodal inputs:** Sensor settings provide images, depth, and point clouds for recognition and localization.
- **Closed-loop integration:** Recognition feeds decisions that send position or velocity targets to the autopilot.
- **Comparison example:** Fix trajectory and controller; vary only occlusion, then compare missed detections and task completion.

Example: e2-UavModeling (Add a 3D model to RflySim3D)

Fix control and motion models; vary scene or sensor conditions to compare perception algorithms.

2.3.2 Model and Algorithm Development: Control Design

Control Algorithms Compute Actuator Commands from Targets and Feedback



① Input Interface; ② Remote Control Processing; ③ Attitude Controller; ④ Motor Control Allocation; ⑤ Output Interface.

- **Inputs and targets:** ① receives remote-control inputs and states; ② converts channels to desired attitude.
- **Attitude control:** ③ computes torque from desired−estimated attitude error via proportional–integral–derivative (PID) control.
- **Allocation and output:** ④ distributes torque and total thrust among motors; ⑤ packages actuator commands, effective after arming.

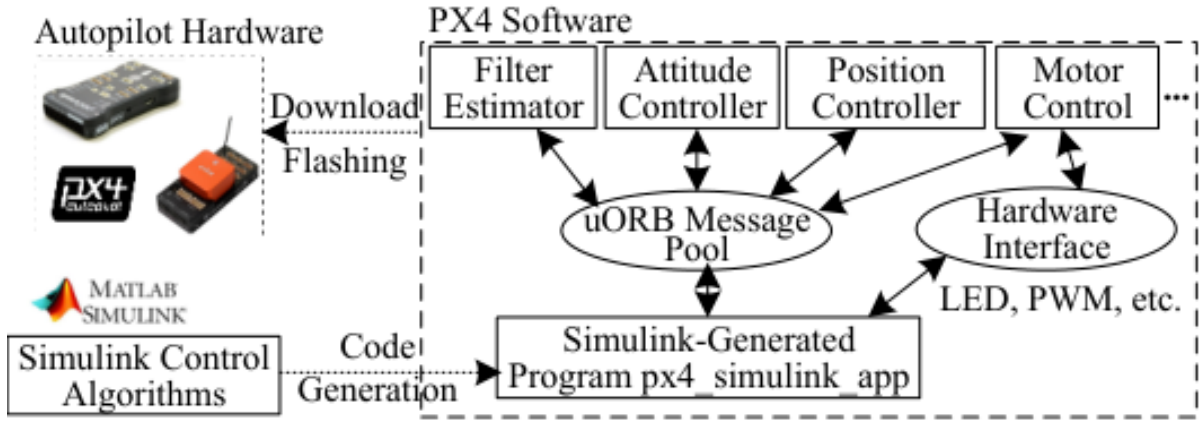
Fig. 2.18(c) Control Algorithm Development and Software Simulation

Example: e5-AttitudeCtrl (Attitude Controller Design Experiment)

Targets enter feedback control; allocation produces commands for individual actuator channels.

2.3.2 Model and Algorithm Development: Program Integration

Integrated Control Programs Join the Autopilot Loop



- **Message connections:** PX4 modules subscribe to inputs and publish results through the micro Object Request Broker (uORB).
- **Target build:** Build generated C/C++ into software or firmware matching the simulation environment or control board.
- **Control handover:** px4_simulink_app takes over actuator output, replacing the original attitude or velocity controller.

Fig. 2.18(d) Automatic Code Generation and Firmware Flashing

Example: e7_Code-Generation (Automatic Flight-Control Firmware Generation with MATLAB)

An algorithm enters the loop only after target integration and assignment of its output responsibility.

Chapter Outline

Architecture and Development Workflows

Chapter 2 Experimental Platform Configuration

2.1 Key Tools

2.2 Installation and First Simulation

2.3 Architecture and Workflows

2.4 API and SDK Interfaces

2.5 Examples and Experimental Routes

2.6 Exercises and Discussion

4 Units in This Section

2.3.1 Overall Architecture and Core Components
Computational roles, feedback data, and observation sources

2.3.2 Model and Algorithm Development
Motion models, control software, and high-level applications

2.3.3 Simulation Modes and Transfer to Real Systems
Software, target hardware, real vehicles, and swarm transfer

2.3.4 Comparative Testing and Results Analysis
Parameter comparisons, fault propagation, and interpretation

Textbook: Sec. 2.2.1–2.2.2 (architecture and components); Sec. 2.3.1–2.3.3 (low-level control, fault testing, perception, and swarms)

2.3.3 Simulation Modes and Transfer to Real Systems: Mode Overview

Simulation Modes Differ in Where Real and Virtual Parts Meet

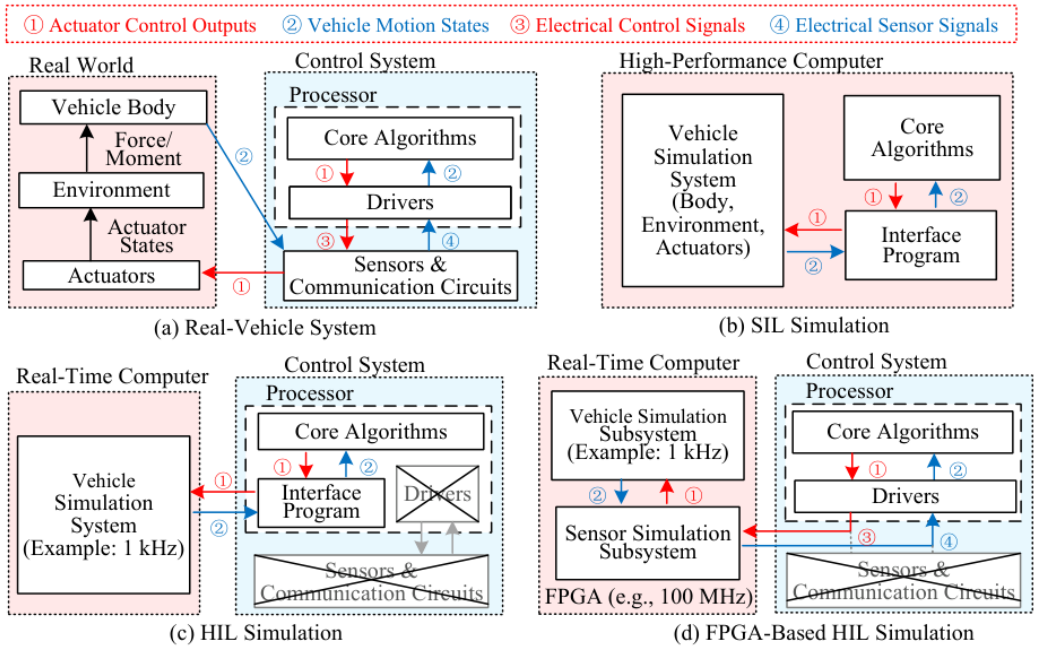


Fig. 2.10 Simulation Modes Supported by RflySim

- **Real-world test:** Physical sensors measure vehicle motion; outputs drive real actuators.
- **SITL:** Control software and the vehicle model run on a computer and exchange network messages.
- **HITL:** PX4 runs on a physical autopilot, exchanging measurements and actuator commands over USB serial.
- **Chip-interface emulation:** A field-programmable gate array (FPGA) emulates sensor chips with matching protocols, signal levels, and timing; drivers remain active.

SITL runs control software; HITL adds target hardware; physical tests add the real vehicle.

2.3.3 Simulation Modes and Transfer to Real Systems: Signal Replacement

HITL and FPGA: The Layer Where Real Signals Are Replaced

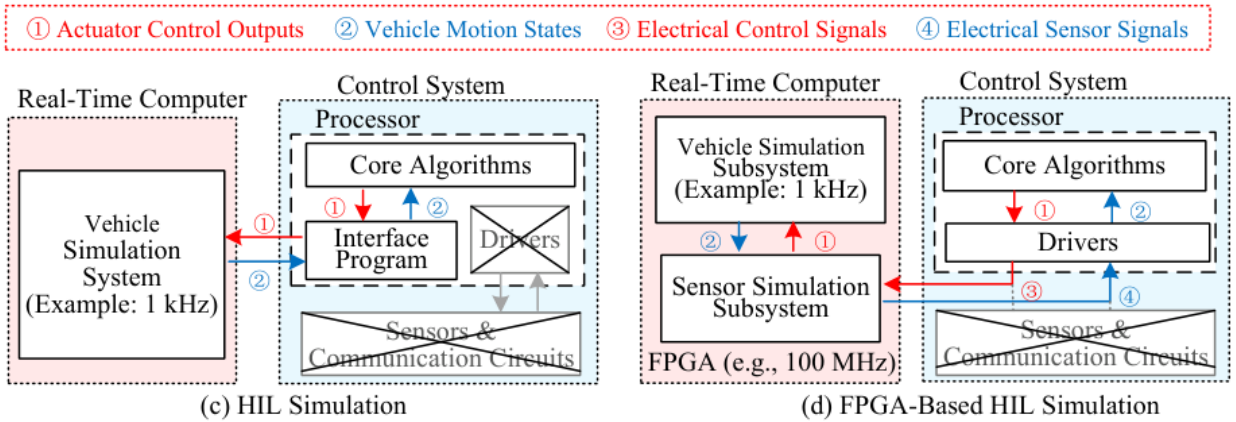


Fig. 2.10(c)(d) Simulation Modes Supported by RflySim (Enlarged Detail)

Reading the figure: ①② are software data; ③④ are electrical signals at pins.
Crossed-out blocks show the physical hardware being replaced.

- **HITL (c):** A forwarding interface bypasses drivers and exchanges ①② with the simulator over USB serial. PX4 natively supports this mode.
- **FPGA approach (d):** Keep real-flight mode and active drivers; replace sensor chips with an FPGA that reads ③ and generates ④. Firmware can stay unchanged after chip-interface adaptation.
- **Cost:** Build device-specific chip models and communication-checking logic.

HITL replaces measurements and actuation in software; FPGA replaces electrical signals at chip pins.

2.3.3 Simulation Modes and Transfer to Real Systems: Signal Replacement

Simulation Modes: Where Real Signals Are Replaced

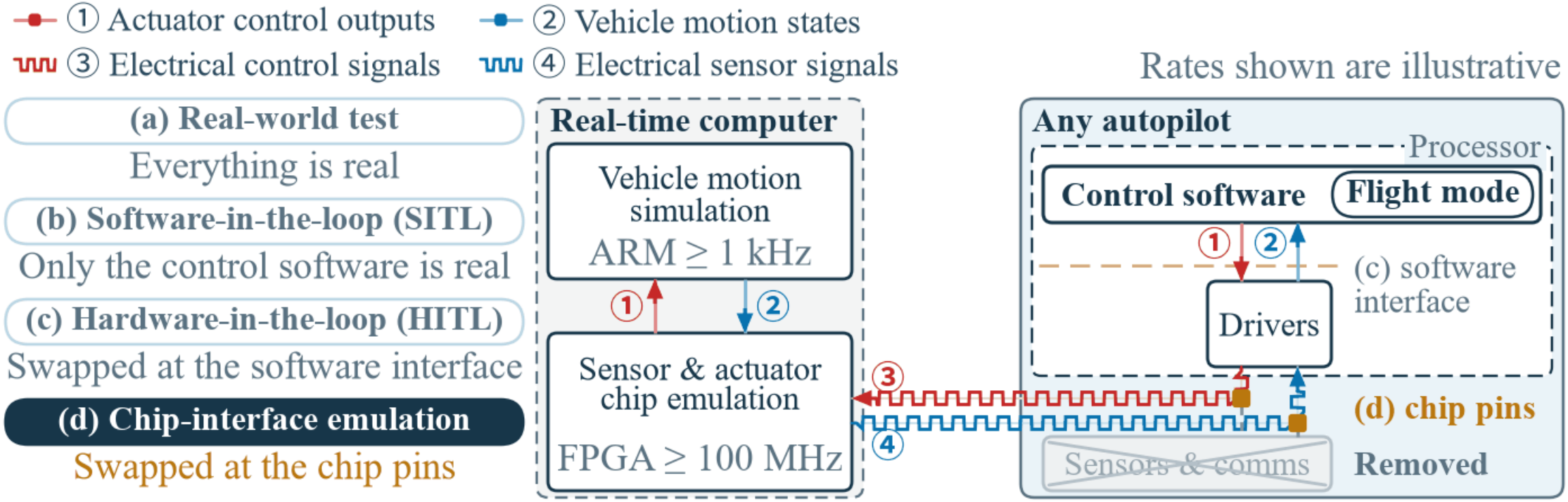


Figure: Replacement Points in Four Modes (Conceptual Diagram Redrawn from Fig. 2.10)

HITL replaces software data; the FPGA approach replaces electrical signals on device pins.

2.3.3 Simulation Modes and Transfer to Real Systems: Mode Selection

Model, Software, and Hardware Tests Validate Different Implementations

Validation Mode	Implementation Tested	Added Validation Coverage
Model-in-the-Loop MIL	Control and vehicle models close a loop in the modeling environment	Control laws, response curves, and numerical solver behavior
Software-in-the-Loop SIL / SITL	Actual control software and vehicle model run on a computer	Mode transitions, module integration, and message exchange
Hardware-in-the-Loop HIL / HITL	Target hardware joins the model loop and receives virtual measurements	Onboard computation cycles, runtime load, and communication timing

SITL and HITL run the same PX4 control logic; processors, timing, and interfaces differ.

Validate control relationships, software integration, and onboard timing at their respective levels.

2.3.3 Simulation Modes and Transfer to Real Systems: Hardware Loop

HITL Closes the Loop with Virtual Measurements and Actuator Commands

Message	Direction and Data	Example Rate
HIL_SENSOR	Model → Autopilot Inertial, barometric, and other measurements	≥250 Hz
HIL_GPS	Model → Autopilot Positioning measurements	≥10 Hz
HIL_ACTUATOR_CONTROLS	Autopilot → Model 16 normalized actuator channels	≥200 Hz
HEARTBEAT	Autopilot → Model System and mode status	1 Hz

Table 2.2 CopterSim HITL Protocol

Measurements feed the autopilot; actuator commands return to the model; heartbeats report runtime status.

2.3.3 Simulation Modes and Transfer to Real Systems: Onboard Deployment

Onboard Programs Can Use Simulated Inputs Before Real Inputs

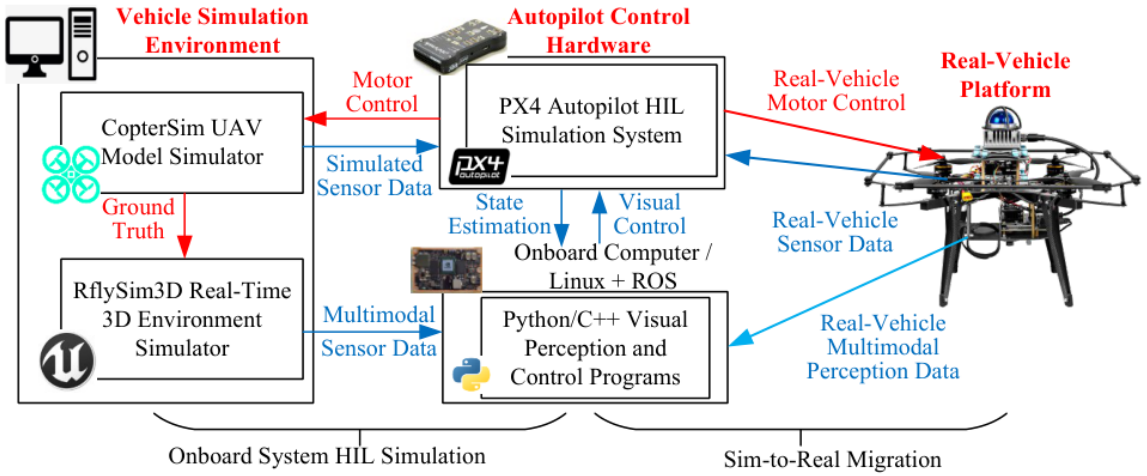


Fig. 2.20 Onboard System HITL Simulation

- **Retain hardware:** Put the autopilot and onboard computer in a simulated loop first; observe the same algorithm's onboard computation and communication.
- **Switch to physical systems:** Replace simulated measurements and images with real sensors; drive motors or control surfaces to reveal noise and latency differences.
- **Deployment requirements:** Match processor architecture, system libraries, and drivers; keep timestamp and coordinate definitions consistent.

Retain hardware and algorithm; replace sensor inputs and the controlled plant to compare transfer effects.

2.3.3 Simulation Modes and Transfer to Real Systems: Physical Validation

Physical Tests Gradually Add Real Sensing, Actuation, and Environments

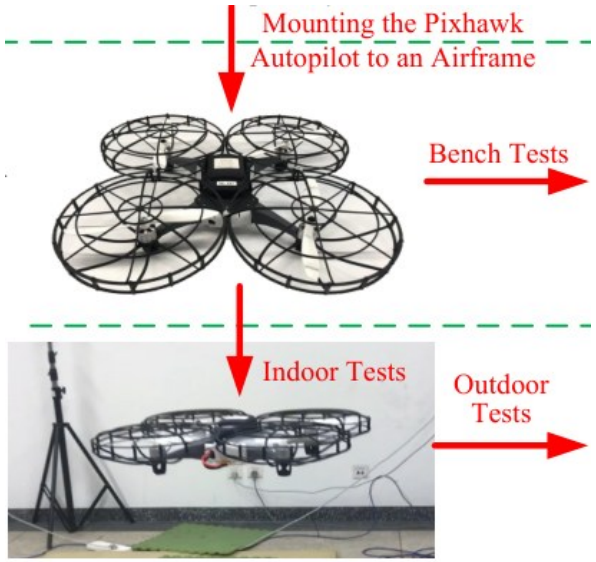


Fig. 2.17 (Detail) Real-Vehicle Experiment Stage



Installation and bench tests: Add motors, control surfaces, and structures; check output direction, actuator response, and vibration.

Tasks and environments: Compare trajectory errors within controlled limits and observe sensor-noise and disturbance effects.

Bench tests expose actuator/structure differences; task tests add environment and motion differences.

2.3.3 Simulation Modes and Transfer to Real Systems: Hybrid Real-Virtual Operation

Hybrid Operation: Some Vehicles Are Virtual, but Control Hardware Is Real

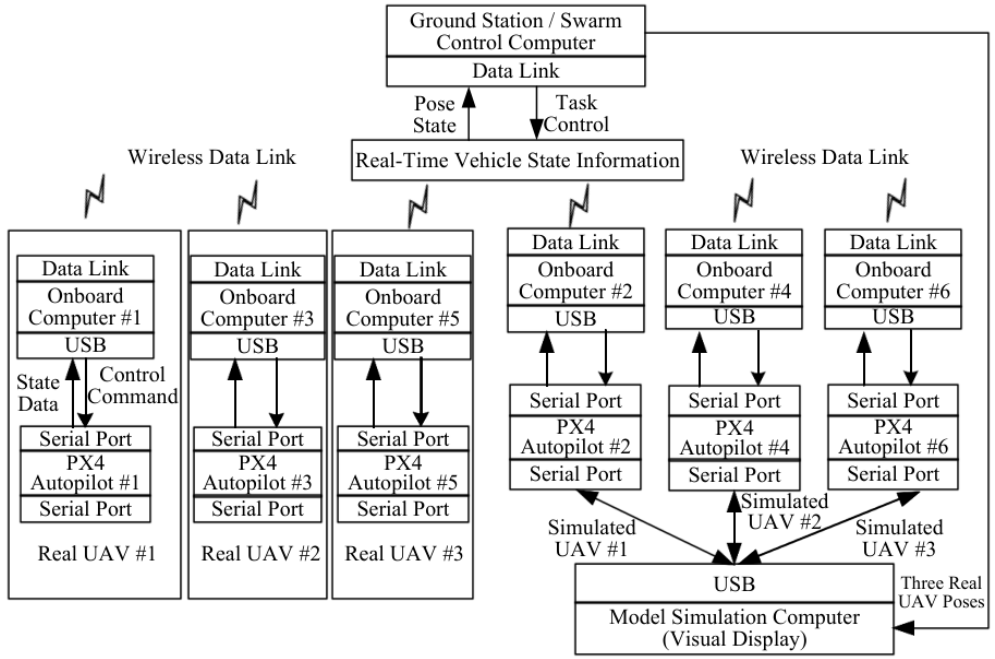


Fig. 2.21 Experimental Architecture for Fixed-Wing HITL Simulation

- **Real hardware:** All six autopilots, onboard computers, and wireless data links are physical devices.
- **Three real, three virtual:** Three aircraft fly; models compute motion and measurements for three virtual aircraft. Real-aircraft poses return via the ground station to the 3D scene.
- **Synchronization and testing:** Synchronize clocks and predict motion to compensate for transmission delay. Virtual aircraft approach real ones to test obstacle avoidance and safety responses in dense formations.

Real and virtual vehicles can share tasks; physical hardware and motion still face real-world constraints.

2.3.3 Simulation Modes and Transfer to Real Systems: Swarm Development Stages

Perception, Decisions, and Swarm Control: Staged Transfer to Real Vehicles



- **Decoupled validation:** Swarm control takes target-pose ground truth from the 3D engine, bypassing image generation and processing. Single-vehicle decisions use detailed image and point-cloud models. Develop both in parallel.
- **Progressive transfer:** Run on real autopilots and onboard computers with simulated data (Fig. 2.20). Then add real vehicles, starting with one real and five virtual; Fig. 2.21 shows three of each.
- **Two physical-test stages:** Check workflows and parameters in stable weather with reliable communication and accurate positioning. Then test safe modes and emergency responses in adverse weather, interference, or injected faults.

Validate swarm control and detailed perception in parallel, then progressively test with real vehicles.

Chapter Outline

Architecture and Development Workflows

Chapter 2 Experimental Platform Configuration

2.1 Key Tools

2.2 Installation and First Simulation

2.3 Architecture and Workflows

2.4 API and SDK Interfaces

2.5 Examples and Experimental Routes

2.6 Exercises and Discussion

4 Units in This Section

2.3.1 Overall Architecture and Core Components
Computational roles, feedback data, and observation sources

2.3.2 Model and Algorithm Development
Motion models, control software, and high-level applications

2.3.3 Simulation Modes and Transfer to Real Systems
Software, target hardware, real vehicles, and swarm transfer

2.3.4 Comparative Testing and Results Analysis
Parameter comparisons, fault propagation, and interpretation

Textbook: Sec. 2.2.1–2.2.2 (architecture and components); Sec. 2.3.1–2.3.3 (low-level control, fault testing, perception, and swarms)

2.3.4 Comparative Testing and Results Analysis: Comparison Design

Parameter Comparisons Begin with a Fixed Baseline

- **Fix the baseline:** Give A and B the same target input, initial state, control parameters, and recording duration so task changes are not mistaken for parameter effects.
- **Define the change:** Change only yaw inertia. Save each model version and its actually loaded parameters; keep the controller, other model parameters, and data sources consistent.
- **Interpret the response:** Record target angular velocity, actual angular velocity, and actuator commands on a common time base. Compare the rise, peak, steady-state error, and convergence.

Keep Baseline A → **Change Only B's Inertia** → **Run the Same Task** → **Compare the Responses**

Run ID: Model version + Actual parameters + Input conditions + Log identify each curve set's run.

Change only the studied component; fix inputs, initial states, and recording conditions to explain responses.

2.3.4 Comparative Testing and Results Analysis: Interpreting Results

Response Curves Reveal the Effects of a Model Change

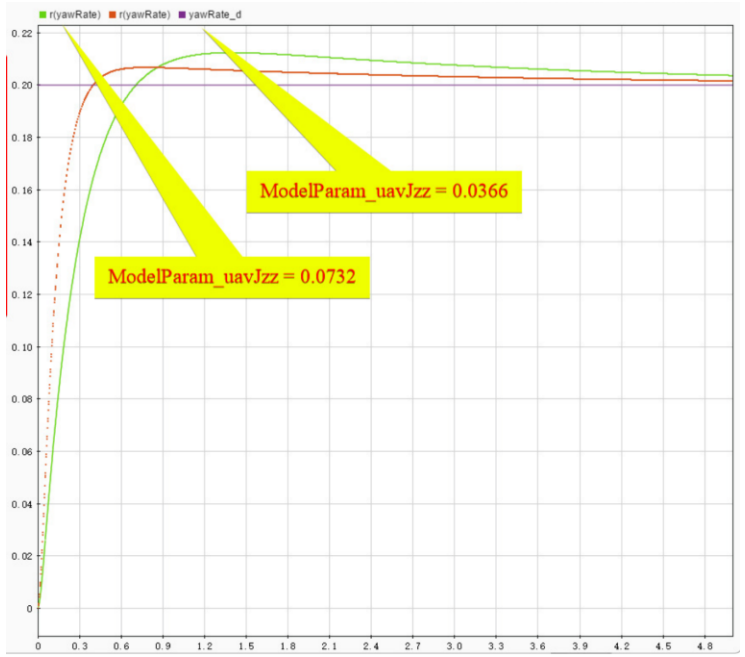


Figure: Yaw Angular-Velocity Comparison in e2-UavModeling

- **Same input:** Both runs target 0.2 rad/s with the same initial state and control parameters, making the responses comparable.
- **Changed inertia:** A (orange) uses 0.0366 kg·m²; B (green) doubles it to 0.0732 kg·m².
- **Curve findings:** B rises more slowly and peaks slightly higher (0.212 vs. 0.207); both approach the target.

Dynamic modeling and parameter comparison: https://rflysim.com/doc/zh/2.Course/ch5/index.html

Compare speed, peak, and error at the same target; conclusions apply only to the tested model and setup.

2.3.4 Comparative Testing and Results Analysis: Reproducible Calculation

Doubling Yaw Inertia Slows the Response at the Same Torque

$$J\dot{\omega} + b\omega = \tau, \quad \tau = b\omega_{\text{ref}}, \quad \omega(t) = \omega_{\text{ref}}(1 - e^{-bt/J})$$

Start from rest; target: 0.2 rad/s; damping: 0.0366 N·m·s/rad; inertia A/B: 0.0366/0.0732 kg·m².

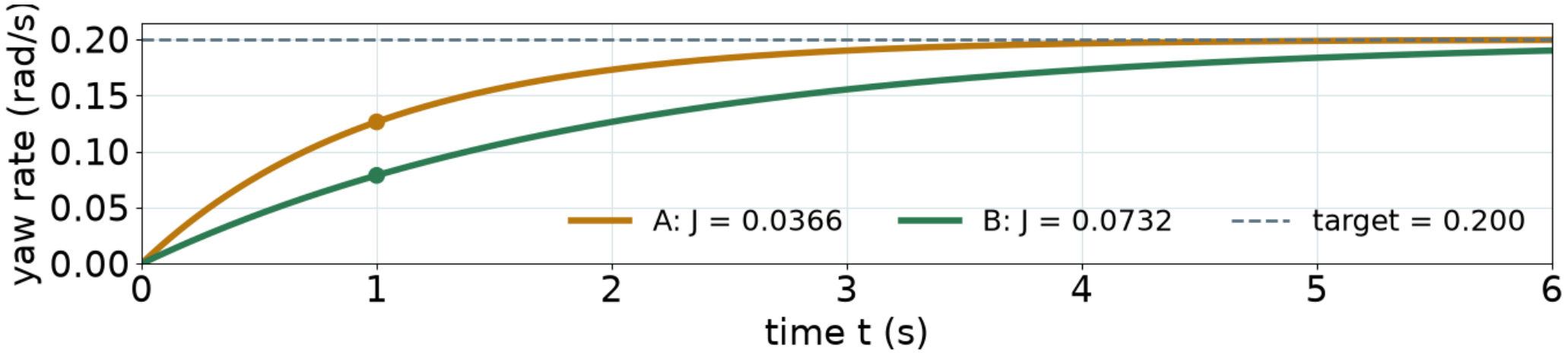


Figure: Yaw Rate at the Same Target and Torque (Simplified Teaching Model; Not Platform Measurements)

At 1 s: A = 0.1264, B = 0.0787 rad/s. No overshoot; this model cannot predict the previous plot's peaks.

With the same target, torque, and damping, doubling inertia increases the time constant from 1 s to 2 s.

2.3.4 Comparative Testing and Results Analysis: Fault Testing

Fault Tests Add Abnormal Conditions to the Same Task

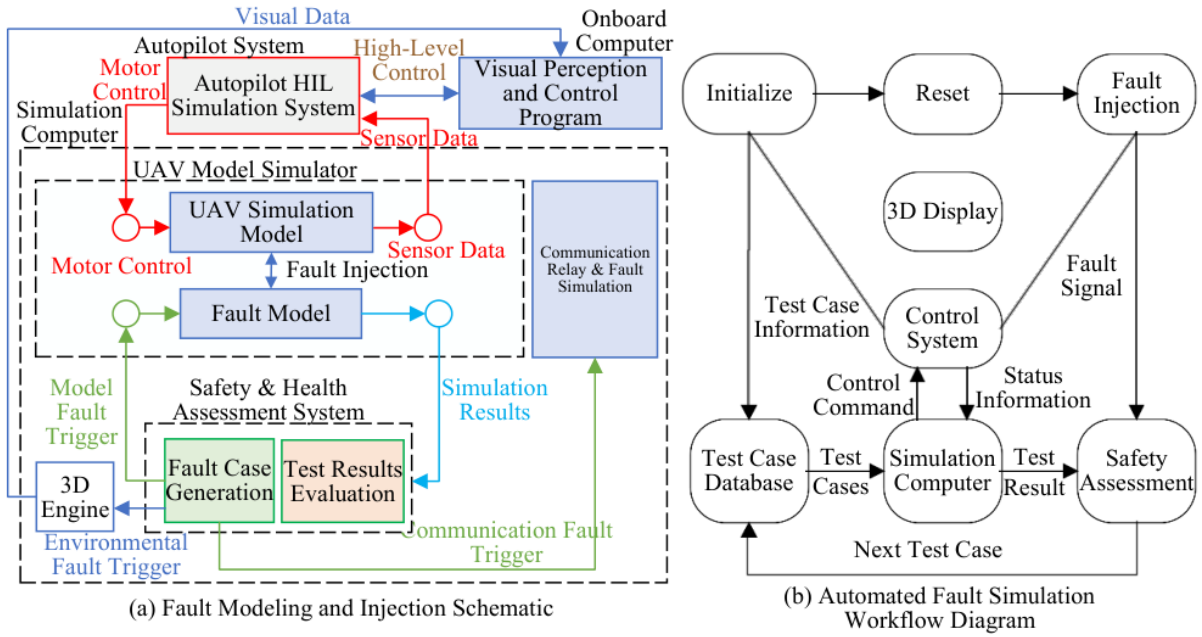


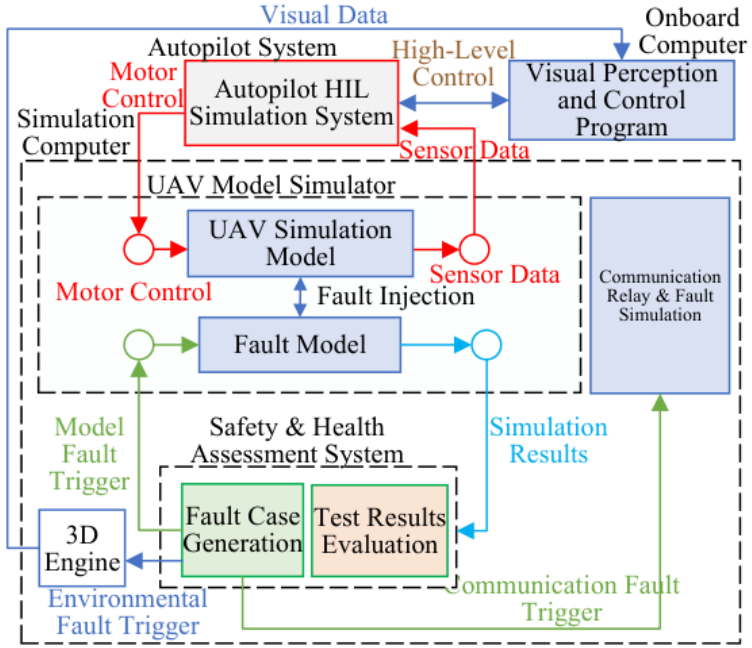
Fig. 2.19 Fault Injection System Design Framework

- **Test purpose:** Check safety mechanisms and fault-tolerant strategies: detection time, diagnosable faults, and response correctness.
- **Case conditions:** Keep a fault-free baseline; define the fault target, onset, magnitude, and duration for the same task.
- **Injection and assessment:** Inject into the model, 3D environment, autopilot, or communication link. Use error, recovery time, and other metrics to classify cases as tolerable, recoverable, or unrecoverable.

Use the same task for normal and fault conditions, then compare errors, detection delay, and recovery.

2.3.4 Comparative Testing and Results Analysis: Fault Propagation

The Injection Point Determines Which Data Changes First



(a) Fault Modeling and Injection Schematic

Fig. 2.19(a) Fault Modeling and Injection Principles

- **Propulsion anomaly:** Inject through the Fault Model. Reduced motor effectiveness first lowers thrust, then changes motion, measurements, and control output.
- **Observation anomaly:** Sensor bias first changes autopilot measurements, then affects motion through state estimation and feedback control.
- **Communication anomaly:** Add delay or packet loss through Communication Relay & Fault Simulation. Log injection time to compare before and after.

Propulsion faults first change forces; observation faults first change measurements; effects follow the loop.

2.3.4 Comparative Testing and Results Analysis: Fault Propagation

The First Signal to Change Reveals Where a Fault Enters the Loop

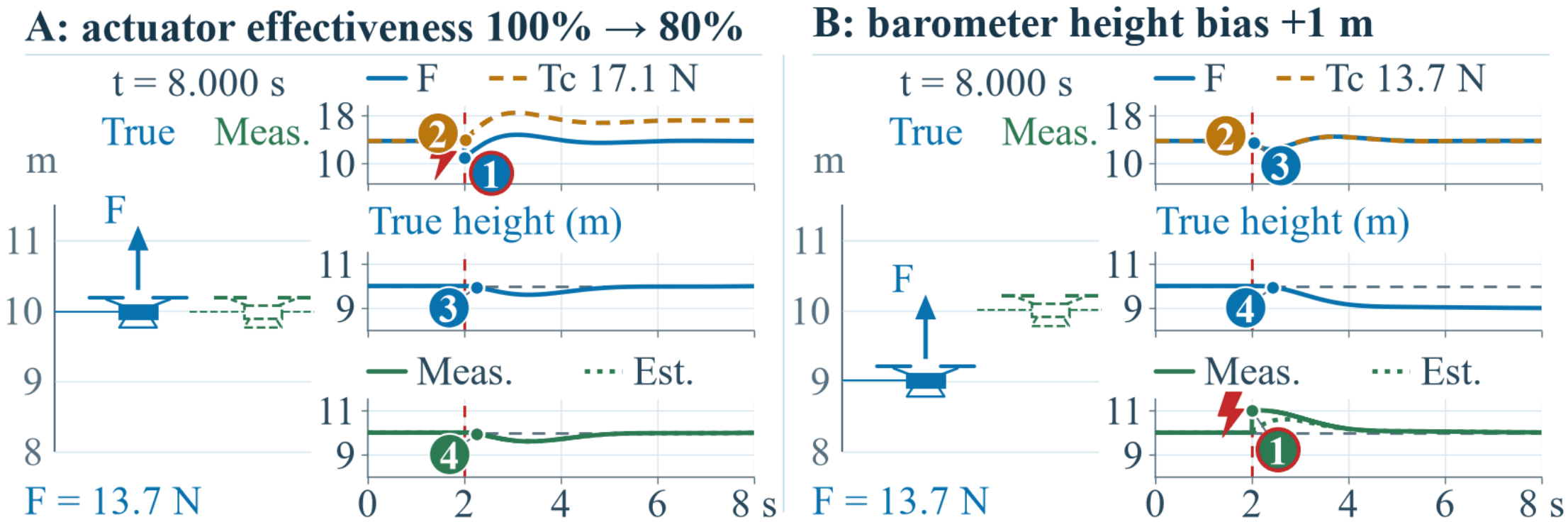
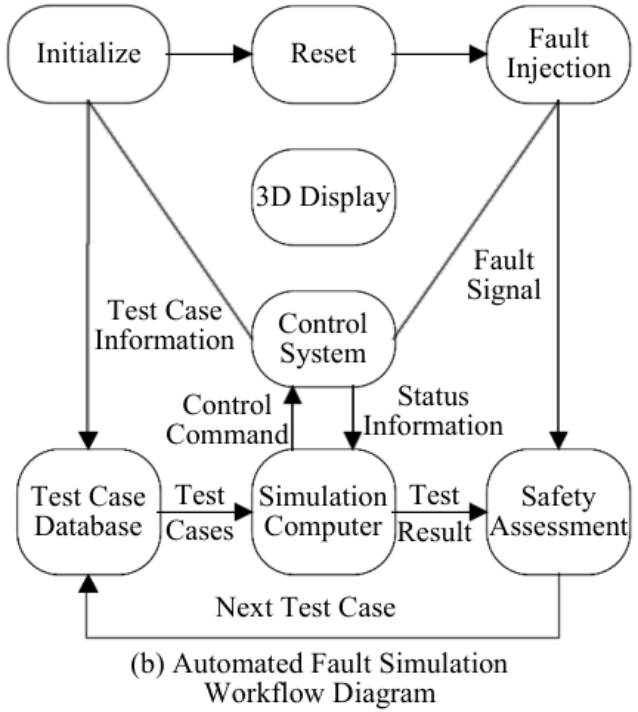


Figure: Order of Signal Changes After Fault Injection (1D Altitude Teaching Simulation)

Reduced actuator effectiveness first changes thrust; barometric bias first changes measured altitude.

2.3.4 Comparative Testing and Results Analysis: Automated Testing

Automated Tests Organize Reset, Execution, Recording, and Assessment



(b) Automated Fault Simulation Workflow Diagram

Fig. 2.19(b) Automated Fault Simulation Workflow

- **Initialize and reset:** Establish software communication, load this round's case, clear residual faults, and restore a stable state such as hover.
- **Inject and assess:** Send fault parameters to the simulation node over UDP. After the test, compute steady-state error, recovery time, and error growth rate.
- **Record and iterate:** Save configuration, logs, and results by round, then load the next case. Count startup failures and safety aborts separately.

Reset the agreed initial state and retain conditions and logs each round for comparable batch results.

2.3.4 Comparative Testing and Results Analysis: Attributing Results

Interpret Results Through Targets, Actuation, and Response

Observed Behavior	Where to Look	Records That Distinguish Causes
Target changes; actuator output does not	Target reception, mode, and control responsibility	Received target, current mode, and control output
Actuator output changes; model response is abnormal	Channels, physical dimensions, and model parameters	Actuator commands; actual parameters; model ground truth
Model moves; display does not match	Instance identity and 3D state mapping	Vehicle ID, transmitted pose, and displayed object
Display is normal; the algorithm gets no new observations	Sensor configuration and data reception chain	Camera ID, frame time, and reception records

Trace from the first changed variable to distinguish faults in model, control, and observation chains.

Exercise: Choose Changes and Validation for a Development Task

- **Change the vehicle:** Replace a quadrotor with a fixed-wing aircraft while retaining the same mission route. List required changes to the motion model, actuator channels, control allocation, and 3D assets, and explain their roles.
- **Change the algorithm:** Keep the existing control and model; replace only visual recognition. Define algorithm inputs and outputs, then compare recognition accuracy and task performance in the same scene.
- **Choose validation:** Measure how long one computation takes on the control board and when its result is transmitted. Choose a test involving the target hardware and specify the times and messages to record.

Answer structure: Changed component → Deliverable → Runtime environment → Comparison data. Explain each choice.

The changed component, runtime artifact, and comparison results must correspond to the same task.

2.3 Architecture and Development Workflows: Exercise Points

Reference Points: Components, Artifacts, Environments, and Data

- **Change the vehicle:** Use a fixed-wing dynamics model. Replace motor-only actuator channels with a motor, ailerons, elevator, and rudder; redesign control allocation for the fixed-wing configuration and use fixed-wing 3D assets. Changing appearance alone leaves the quadrotor's physical response.
- **Change the algorithm:** Input camera images; output a target location or motion target. Compare recognition accuracy offline on identical images. In closed-loop tests, fix the scene, initial state, model, and controller, then observe trajectories and task completion.
- **Choose validation:** Use HITL with the target control board. Record computation start/end times, message transmission/reception timestamps, and processor load.

2.3 Architecture and Development Workflows: Lesson Summary

Architecture, Development, and Testing

- **Architecture:** The autopilot estimates and controls, the model computes motion, the scene generates observations, and high-level programs provide task targets. Data exchange closes the loop.
- **Deliverables:** Motion models become DLLs, low-level algorithms enter control software or firmware, and perception and decision-making become high-level applications. Each artifact has a defined loading and runtime location.
- **Validation:** Model, software, and hardware tests examine control relationships, program interaction, and onboard timing, respectively. Physical tests add real sensing, actuation, and environments.
- **Results analysis:** Fix inputs and initial states, retain actual parameters and logs, and trace changes through targets, actuator commands, motion, and observations to explain why a change has its observed effect.

Next: 2.4 API and SDK Interfaces → 2.5 Examples and Experimental Routes

Trace data through module inputs and outputs to connect interface calls with experimental observations.