Specification and implementation code for Lane Change Control (to safe maneuvering into adjacent lane) in an Autonomous Driving System (ADS):

Nice — lane-change control is a safety-critical maneuver that needs a clear spec + a robust supervisor that coordinates perception, prediction, planning (trajectory), longitudinal & lateral controllers, and monitors for aborts. Below I give a practical specification and a single-file Python reference you can run in simulation. It implements a state-machine supervisor (Prepare → Execute → Complete / Abort), a quintic lateral trajectory generator blended with a longitudinal distance/speed profile, and an example controller stack (pure-pursuit lateral + PID longitudinal) plus safety checks (gap, TTC, blind-spot, max curvature, watchdog).

I designed this to be readable, modular, and ready to adapt to your vehicle API / ROS topics.

Specification — Lane Change Control (Safe Adjacent-Lane Maneuver)

Purpose

Perform a safe lateral transfer from the current lane to an adjacent target lane while maintaining safety (no collisions), comfort (bounded accel/jerk), and regulatory constraints.

Modes / States

- Idle — no lane change requested.
- Prepare — check preconditions (gap, speed, indicator, planner permits), reserve lane, compute candidate trajectory.
- Execute — follow planned trajectory while continuously checking safety constraints.

- Complete — successfully merged into target lane and clear of origin lane.
- Abort — abort maneuver (return to lane center or safe stop) due to safety violation or external request.

Interface

Inputs:

- state: ego {x,y,yaw,v,a,delta}.
- lane_current, lane_target: lane center lines or polynomials.
- perception: list of other actors with {x,y,vx,vy, bbox} relative to world; blind-spot flags.
- planner_req: user/planner request (left/right lane change).
- dt: control timestep.

Outputs:

- actuator commands: throttle/brake and delta_cmd (steering).
- status: state_machine_state, progress (0..1), abort_reason if any.

Preconditions (for transition Idle→Prepare)

- Lane-change request accepted by high-level planner (policy).
- Ego speed within allowed range v_min ≤ v ≤ v_max.
- Safe longitudinal gap in target lane ahead & behind: d_ahead ≥ d_ahead_min(v) and d_behind ≥ d_behind_min(v) (dynamic).
- No fast-approaching vehicle in blind spot (TTC > ttc_min, or predicted min distance > safety_margin).
- Lane change feasible wrt curvature & road

geometry.

Trajectory Generation

- Generate a smooth lateral displacement y(s) vs longitudinal s across lane width using a quintic polynomial or a spline that enforces: start/end lateral pos, start/end lateral velocity=0, start/end lateral accel=0 (smooth).
- Longitudinal profile: maintain speed v_ref or adjust (slightly accelerate/decelerate) to fit gap and comfort constraints. Optionally use time-optimal blending subject to a_max, jerk_max.
- Combine to parametric trajectory (x(t), y(t)) or (s(t), y(s)).

Execution & Monitoring

- Follow trajectory with lateral controller + longitudinal controller.

- Continuously predict other actors over short horizon; if predicted conflict (distance < safety_margin or TTC < ttc_abort) then Abort or Yield depending on severity:
    - Mild: slow down and re-evaluate (soft abort).
    - Severe: immediate abort — return to lane center or emergency brake depending on proximity.
- Enforce curvature & lateral accel: a_lat = v^2 * kappa ≤ a_lat_max.
- Enforce actuator limits and rate limits.
- On completion (rear axle fully inside target lane and lane-clear for a margin) → set Complete.

Abort behaviour

- If abort during Prepare: cancel and stay in current lane.

- If abort during Execute:
  - If safe to return: compute return trajectory back to original lane and enact it.
  - If imminent collision: emergency braking and minimize risk.
- Log abort reason & require re-request to try again.

Guarantees

- No actuator command requests beyond vehicle limits.
- Safety monitors run at controller frequency; worst-case reaction enforced by emergency stop routine.
- Mutual exclusion with other maneuvers (no parallel lane-change requests).

Reference Implementation (Python)

- Run as a toy sim: kinematic bicycle plant + simple actor predictions.
- Implements: Supervisor + Quintic lateral trajectory generator + Longitudinal profile + Pure Pursuit lateral controller + PID longitudinal controller + monitors (gap, TTC).

"""

lane_change.py — reference lane-change supervisor + trajectory + controllers.

Run as a toy simulation / starting point. Not production-ready for road tests.

"""

```
import math
import copy
from dataclasses import dataclass
from typing import List, Tuple, Optional


# -----------------------
# Utility helpers
# -----------------------
def clamp(x, lo, hi):
    return hi if x > hi else lo if x < lo else x


def normalize_angle(a):
    return (a + math.pi) % (2*math.pi) - math.pi


# simple 2D distance
```

```
def dist(a, b):
    return math.hypot(a[0]-b[0], a[1]-b[1])


# -----------------------
# Vehicle state / actor
# -----------------------
@dataclass
class VehicleState:
    x: float
    y: float
    yaw: float
    v: float
    a: float = 0.0
    delta: float = 0.0
```

```
@dataclass
class Actor:
    x: float
    y: float
    vx: float
    vy: float
    id: int

    def predict(self, dt: float, steps: int) ->
List[Tuple[float,float]]:
        """Simple constant-velocity prediction."""
        preds = []
        for s in range(1, steps+1):
            t = s * dt
            preds.append((self.x + self.vx*t, self.y +
```

```
self.vy*t))
        return preds

# ----------------------
# Quintic polynomial generator for lateral
displacement y(s)
# We parameterize lateral offset as function of
longitudinal progress s in [0, S]
# Boundary conditions: y(0)=y0, y'(0)=0, y''(0)=0 ;
y(S)=yS, y'(S)=0, y''(S)=0
# Solve coefficients for y(s)=a0 + a1 s + a2 s^2 + a3
s^3 + a4 s^4 + a5 s^5
# ----------------------
def quintic_coeffs(s0, s1, y0, y1):
    # s0=0, we assume s0==0 for simplicity
```

```
S = s1
# Standard quintic with zero derivatives at
endpoints:
# a0=y0; a1=0; a2=0; solve for a3,a4,a5 using:
# y(S)=a0 + a1 S + a2 S^2 + a3 S^3 + a4 S^4 +
a5 S^5 = y1
# y'(S)=a1 + 2 a2 S + 3 a3 S^2 + 4 a4 S^3 + 5 a5
S^4 = 0
# y''(S)=2 a2 + 6 a3 S + 12 a4 S^2 + 20 a5 S^3 =
0
# with a1=a2=0
a0 = y0
a1 = 0.0
a2 = 0.0
# Solve linear system for a3,a4,a5
```

```
    # [ S^3    S^4    S^5 ] [a3] = [y1 - a0]
    # [3S^2 4S^3 5S^4] [a4] = [0]
    # [6S    12S^2 20S^3] [a5] = [0]
    # Solve directly (closed-form for zero
derivatives):
    # Using standard formulas:
    d = y1 - a0
    a3 = (10 * d) / (S**3)
    a4 = (-15 * d) / (S**4)
    a5 = (6 * d) / (S**5)
    return (a0, a1, a2, a3, a4, a5)


def quintic_eval(coeffs, s):
    a0,a1,a2,a3,a4,a5 = coeffs
    y = a0 + a1*s + a2*s*s + a3*s**3 + a4*s**4 +
```

```
a5*s**5
        dy = a1 + 2*a2*s + 3*a3*s*s + 4*a4*s**3 +
5*a5*s**4
        ddy = 2*a2 + 6*a3*s + 12*a4*s*s + 20*a5*s**3
        return y, dy, ddy


# -----------------------
# Simple longitudinal PID controller (re-usable)
# -----------------------
@dataclass
class PID:
        kp: float; ki: float; kd: float
        i_clamp: float = 1.0
        integ: float = 0.0
        prev_v: Optional[float] = None
```

```
    def step(self, v_ref, v, dt):
        e = v_ref - v
        self.integ = clamp(self.integ + e * dt, -
self.i_clamp, self.i_clamp)
        dv = 0.0 if self.prev_v is None else (v -
self.prev_v)/max(dt,1e-6)
        self.prev_v = v
        return self.kp*e + self.ki*self.integ -
self.kd*dv


# ----------------------
# Pure Pursuit lateral controller (simple)
# ----------------------
class PurePursuit:
```

```
    def __init__(self, wheelbase=2.7,
lookahead_base=6.0, lookahead_gain=0.5,
steer_max=math.radians(30)):
        self.wheelbase = wheelbase
        self.lookahead_base = lookahead_base
        self.lookahead_gain = lookahead_gain
        self.steer_max = steer_max

    def lookahead(self, v):
        return max(1.0, self.lookahead_base +
self.lookahead_gain*v)

    def find_target_in_traj(self, traj_xy, state:
VehicleState, Ld):
        # traj_xy: list of (x,y) in world frame
```

```
        for px,py in traj_xy:
            # transform point into vehicle frame
            dx = px - state.x; dy = py - state.y
            x_v = dx*math.cos(-state.yaw) -
dy*math.sin(-state.yaw)
            y_v = dx*math.sin(-state.yaw) +
dy*math.cos(-state.yaw)
            if x_v > 0 and math.hypot(x_v,y_v) >=
Ld:
                return x_v, y_v
        # fallback to last point
        px,py = traj_xy[-1]
        dx = px - state.x; dy = py - state.y
        x_v = dx*math.cos(-state.yaw) -
dy*math.sin(-state.yaw)
```

```
        y_v = dx*math.sin(-state.yaw) +
dy*math.cos(-state.yaw)
        return x_v,y_v


    def control(self, traj_xy, state: VehicleState):
        Ld = self.lookahead(state.v)
        xld, yld = self.find_target_in_traj(traj_xy,
state, Ld)
        if abs(xld) < 1e-6 and abs(yld) < 1e-6:
            return 0.0
        alpha = math.atan2(yld, xld)
        kappa =
2.0*math.sin(alpha)/(max(math.hypot(xld,yld), 1e-
6))
        delta = math.atan(self.wheelbase * kappa)
```

```
        return clamp(delta, -self.steer_max,
self.steer_max)


# ----------------------

# Supervisor for lane change

# ----------------------

@dataclass

class LaneChangeConfig:

    # distance/time margins

    d_ahead_min: float = 20.0       # required gap
ahead in target lane (m)

    d_behind_min: float = 10.0      # required gap
behind in target lane (m)

    ttc_min: float = 2.0            # minimal TTC
to consider safe (s)
```

```
    v_min: float = 3.0                    # min speed to attempt lane change
    v_max: float = 35.0                   # max speed allowed for lane change
    S_nominal: float = 30.0               # nominal longitudinal distance to execute lane change (m)
    max_lat_acc: float = 2.5              # lateral accel allowed (m/s^2)
    a_max: float = 2.0
    a_min: float = -6.0
    jerk_max: float = 2.0


class LaneChangeSupervisor:
    def __init__(self, cfg: LaneChangeConfig):
        self.cfg = cfg
```

```
        self.state = "IDLE"
        self.progress = 0.0
        self.traj_xy = []              # planned XY
trajectory (world coords)
        self.traj_s = []               # param s
(longitudinal progress)
        self.traj_y = []               # lateral offsets
        self.S = 0.0
        self.t_total = 0.0
        self.t_elapsed = 0.0
        self.direction = 0             # -1 left, +1 right
        self.abort_reason = None

    def check_gaps(self, ego: VehicleState, actors:
List[Actor], direction: int) -> bool:
```

```
        """
        Check required gaps in target lane. Actors
predicted in target lane:
        - require forward gap >= d_ahead_min and
rear gap >= d_behind_min
        """
        # For toy sim, just compute nearest actor
ahead/behind in target lane lateral band:
        ahead_min = float('inf')
        behind_min = float('inf')
        for a in actors:
            # compute relative along-track distance
(project onto ego heading)
            dx = a.x - ego.x; dy = a.y - ego.y
            s_rel = dx*math.cos(ego.yaw) +
dy*math.sin(ego.yaw)
```

```
            # lateral offset
            l_rel = -dx*math.sin(ego.yaw) +
dy*math.cos(ego.yaw)   # left positive
            # target lane is direction: left=-1 means
target lateral offset negative/positive depending on
conv.
            # approximate: look in lateral band ~
lane_width/2 around target lane center
            if abs(l_rel) < 4.0:   # arbitrary lateral
threshold for "in lane"
                if s_rel >= 0:
                    ahead_min = min(ahead_min,
s_rel)
                else:
                    behind_min =
min(behind_min, -s_rel)
```

```
            # If no actor detected, treat as large gap
            if math.isinf(ahead_min): ahead_min = 1e6
            if math.isinf(behind_min): behind_min =
1e6
            ok = (ahead_min >= self.cfg.d_ahead_min)
and (behind_min >= self.cfg.d_behind_min)
            return ok

      def compute_required_S(self, ego: VehicleState):
            # nominal longitudinal distance S, can be
adapted by speed
            # faster speed -> need more distance/time
            return max(10.0, self.cfg.S_nominal * (ego.v
/ 10.0))
```

```
    def plan_trajectory(self, ego: VehicleState, lane_offset: float, direction: int):
        """
        Plan a combined longitudinal s ∈ [0,S] and lateral y(s) quintic offset from current lane center to target lane center.
        lane_offset: lateral displacement from current lane center to target lane center (m) (positive left)
        direction: -1 left, +1 right
        """
        self.direction = direction
        S = self.compute_required_S(ego)
        self.S = S
        coeffs = quintic_coeffs(0.0, S, 0.0, lane_offset)    # lateral offsets y(0)=0 ->
```

y(S)=lane_offset

sample trajectory in s (longitudinal) and convert to world XY using ego heading as reference

N = int(max(50, S/0.5))

traj_xy = []

s_list = []

y_list = []

for i in range(N+1):

s = S * i / N

y, dy, ddy = quintic_eval(coeffs, s)

world position: start at ego rear-axle position, move forward by s along yaw, offset laterally by y

x_w = ego.x + s * math.cos(ego.yaw) - y * math.sin(ego.yaw)

```
                y_w = ego.y + s * math.sin(ego.yaw) +
y * math.cos(ego.yaw)

                traj_xy.append((x_w, y_w))

                s_list.append(s)

                y_list.append(y)

            # time allocation: choose time to execute
respecting a_max, jerk_max heuristics

            # simple: t_total = S / v_nom where v_nom
= ego.v (or min speed to be safe)

            v_nom = max(3.0, ego.v)

            t_total = S / v_nom

            self.traj_xy = traj_xy

            self.traj_s = s_list

            self.traj_y = y_list

            self.t_total = max(1.0, t_total)
```

```
        self.t_elapsed = 0.0
        return True

    def safety_monitor_conflict(self, ego:
VehicleState, actors: List[Actor], dt: float) ->
Optional[str]:
        """
        Predict actors for short horizon and detect
conflicts with planned traj.
        Return abort reason string if conflict
detected (TTC < threshold etc).
        """
        # predict actors for horizon equal to
remaining time up to t_total
        steps = int(max(10, (self.t_total -
self.t_elapsed)/dt))
```

```
        steps = min(50, steps)
        for a in actors:
            preds = a.predict(dt, steps)
            # check minimal distance to any
waypoint in remaining traj
            for idx, (tx,ty) in
enumerate(self.traj_xy):
                # map trajectory index to time:
t_idx ~ idx/N * t_total
                # we compare predicted actor
position at similar time
                t_idx = (idx /
max(1,len(self.traj_xy)-1)) * self.t_total
                pred_idx = int(min(steps-1, max(0,
round(t_idx / dt) - 1)))
                px,py = preds[pred_idx]
```

```
                d = math.hypot(px - tx, py - ty)
                if d < 3.0:   # safety margin (m)
                    # compute ttc approx w.r.t ego
along-line if closing fast
                    rel_v = math.hypot(a.vx, a.vy)
- ego.v
                    if rel_v > 0:
                        # naive TTC
                        ttc = d / rel_v if rel_v>1e-
3 else float('inf')
                        if ttc < self.cfg.ttc_min:
                            return
f"CONFLICT_TTC_{a.id}"
                    else:
                        return
f"POTENTIAL_CONFLICT_CLOSE_{a.id}"
```

```
        return None

    def step(self, ego: VehicleState, actors:
List[Actor], request: Optional[str], dt: float,
             lane_offset=3.5):
        """
        request: "left" / "right" or None
        lane_offset: typical lane width (positive
means left)
        """
        # state machine
        if self.state == "IDLE":
            if request in ("left","right"):
                # check speed constraints
                if ego.v < self.cfg.v_min or ego.v >
```

```
self.cfg.v_max:
                    self.abort_reason =
"SPEED_NOT_ALLOWED"
                    self.state = "IDLE"
                    return {"state": self.state}
                direction = -1 if request=="left"
else 1
                # check static gaps first
                if not self.check_gaps(ego, actors,
direction):
                    self.abort_reason =
"GAP_NOT_SAFE"
                    return {"state":"IDLE",
"abort":self.abort_reason}
                # plan
                ok = self.plan_trajectory(ego,
```

```
lane_offset*direction, direction)
                if not ok:
                    self.abort_reason =
"PLAN_FAILED"
                    return {"state":"IDLE",
"abort":self.abort_reason}
                self.state = "PREPARE"
                self.abort_reason = None
                return {"state":self.state}
            else:
                return {"state":"IDLE"}

        elif self.state == "PREPARE":
            # re-check dynamic gaps / last-moment
actors
```

```
            if not self.check_gaps(ego, actors,
self.direction):
                self.state = "IDLE"
                self.abort_reason = "GAP_LOST"
                return {"state":self.state,
"abort":self.abort_reason}
            # small final monitors (blind spot):
check close actors in lateral band
            for a in actors:
                dx = a.x - ego.x; dy = a.y - ego.y
                s_rel = dx*math.cos(ego.yaw) +
dy*math.sin(ego.yaw)
                l_rel = -dx*math.sin(ego.yaw) +
dy*math.cos(ego.yaw)
                if abs(l_rel) < 2.0 and abs(s_rel) <
5.0:
```

```
                    self.state = "IDLE"
                    self.abort_reason = 
"BLINDSPOT_OCCUPIED"
                    return {"state":self.state, 
"abort":self.abort_reason}
            # pass -> execute
            self.state = "EXECUTE"
            self.t_elapsed = 0.0
            return {"state":self.state}

        elif self.state == "EXECUTE":
            # monitor predicted conflicts along 
remaining trajectory
            abort = 
self.safety_monitor_conflict(ego, actors, dt)
```

```
        if abort:
            self.state = "ABORT"
            self.abort_reason = abort
            return {"state":self.state,
"abort":self.abort_reason}
        # monitor lateral accel limit:
approximate curvature at current s
        # find index corresponding to progress
        N = len(self.traj_s)
        if N <= 1:
            idx = -1
        else:
            frac = min(1.0, self.t_elapsed /
max(1e-3, self.t_total))
            idx = int(frac*(N-1))
```

```
            # compute approximate curvature from
y'' and y'
            # For simplicity we compute using local
s->y polynomial if available. Here skip detailed
curvature.
            # Progress time
            self.t_elapsed += dt
            self.progress = min(1.0, self.t_elapsed /
max(1e-6, self.t_total))
            if self.progress >= 1.0:
                self.state = "COMPLETE"
                return {"state":self.state}
            return {"state":"EXECUTE",
"progress":self.progress}

        elif self.state == "ABORT":
```

```
                # try to return to original lane smoothly:
here we simply set state to IDLE (toy)
                # Real system: plan return trajectory
and execute
                # For now mark aborted and go IDLE
                reason = self.abort_reason
                self.state = "IDLE"
                return {"state":"IDLE", "abort":reason}

            elif self.state == "COMPLETE":
                # finished; hold lane and clear
                self.state = "IDLE"
                return {"state":"IDLE",
"complete":True}
```

```
        else:
            self.state = "IDLE"
            return {"state":"IDLE"}

# ----------------------
# Toy plant & controller loop for demo
# ----------------------
def kinematic_bicycle_step(state: VehicleState,
a_cmd: float, delta_cmd: float,
                                  wheelbase: float, dt:
float, steer_rate_max=math.radians(30)):
    # rate-limit steering
    max_step = steer_rate_max * dt
    delta = clamp(delta_cmd, state.delta - max_step,
state.delta + max_step)
```

```
        delta = clamp(delta, -math.radians(40),
math.radians(40))

        yaw_rate = (state.v / wheelbase) *
math.tan(delta)

        state.x += state.v * math.cos(state.yaw) * dt

        state.y += state.v * math.sin(state.yaw) * dt

        state.yaw = normalize_angle(state.yaw +
yaw_rate * dt)

        state.v = max(0.0, state.v + a_cmd * dt)

        state.a = a_cmd

        state.delta = delta

        return state


# ----------------------

# Demo usage: simulate a lane change request with a
```

slow actor in target lane

```
# ----------------------

if __name__ == "__main__":

    dt = 0.05

    ego = VehicleState(x=0.0, y=0.0, yaw=0.0,
v=20.0/3.6)   # 20 km/h (~5.56 m/s)

    # actor ahead in target lane (right lane) far
enough

    actors = [Actor(x=50.0, y=-3.5, vx=0.0, vy=0.0,
id=1),

              Actor(x=-30.0, y=-3.5, vx=5.0,
vy=0.0, id=2)]

    sup_cfg = LaneChangeConfig()

    supervisor = LaneChangeSupervisor(sup_cfg)

    # controllers
```

```
    pp = PurePursuit()

    pid = PID(kp=0.6, ki=0.1, kd=0.02)

    # request right lane change

    request = "right"

    log = []

    for step in range(800):

        t = step*dt

        out_sup = supervisor.step(ego, actors,
request if step==0 else None, dt, lane_offset= -3.5)
# right = negative lateral in our convention

        # if executing, get target traj and follow it:

        if supervisor.state == "EXECUTE":

            # follow preplanned trajectory: sample
nearest remaining waypoint

            # choose waypoint index proportional to
```

```
progress
            idx = int(min(len(supervisor.traj_xy)-1,
round(supervisor.progress * (len(supervisor.traj_xy)-
1))))
            traj_remain = supervisor.traj_xy[idx:]
            delta_cmd = pp.control(traj_remain,
ego)
            # simple longitudinal target: keep same
speed
            v_target = 20.0/3.6
            a_cmd = pid.step(v_target, ego.v, dt)
            # apply actuator limits (toy)
            a_cmd = clamp(a_cmd, sup_cfg.a_min,
sup_cfg.a_max)
            # step plant
            ego = kinematic_bicycle_step(ego,
```

```
a_cmd, delta_cmd, wheelbase=2.7, dt=dt)
        else:
            # maintain lane center (no lane change)
            # small steering to stay
            delta_cmd = 0.0
            a_cmd = 0.0
            ego = kinematic_bicycle_step(ego,
a_cmd, delta_cmd, wheelbase=2.7, dt=dt)
        log.append((t, copy.copy(ego),
supervisor.state, out_sup))
        if step % 50 == 0:
            print(f"t={t:.2f}s
sup_state={supervisor.state}
progress={supervisor.progress:.2f}")
        if supervisor.state == "IDLE" and
out_sup.get("complete", False):
```

```
        print("Lane change complete.")
        break
print("final ego:", ego)
```

Tuning & Practical Considerations

1. Gap & TTC rules — use conservative thresholds first. Typical start: d_ahead_min ≈ 20–30 m, d_behind_min ≈ 10 m, ttc_min ≈ 2–3 s. Make these speed-dependent.
2. Trajectory length S — adapt S with speed: higher speeds need more longitudinal distance/time; allow planner to adjust with traffic.
3. Polish the longitudinal profile — instead of constant speed, allow gentle decel/accel to fit gaps (cooperative gap closing).
4. Predict other actors — constant-velocity is minimal; better: use intent/policy-aware

prediction (cut-in probabilities), use probabilistic safety checks.

5. Abort strategy — returning to origin lane is often safer than hard braking if space permits. Implement return trajectory generator symmetric to lane-change one.
6. Indicator & human factors — if human override exists, provide indicator and audible warnings; if shared-autonomy, respect handback.
7. Integration — this supervisor must be tightly integrated with perception (lane confidences, blindspot sensors), motion planner (for feasibility), and vehicle CAN interface for actuator limits and state.
8. Validation — simulate many cut-in scenarios, run HIL, and closed-course validation. Log near-miss metrics, TTC distributions, and yaw/lat-accel

peaks.