Project Ideas

A menu for your final project



Your final project is built in a group of three over about three weeks and shown in a 7-minute demo in Session XIII. The ideas on this page are a menu to choose from. **Your own idea is the best choice**: pitch it to me, and you build it once I approve it.

You build with your agent, so writing the code is the smaller part of the work. The larger part is what the agent cannot do for you: collecting data, testing against answers you know, balancing a game, and defending your decisions at the demo.

Every idea has the same parts: what the class sees at the demo, the core that must work, a stretch if time allows, the work for three, the data and tools, and what to watch out for.

Games you can play at the demo

These five are built to be played on the projector by your classmates, so the rules, the pace and the scoring are the product. Your main job besides directing the agent is playtesting: tuning the numbers until the game is fun, fair and over in a few minutes.

1. The Beer Game, online

The supply chain classic as a web game: four players order beer up the chain, delays do the rest, and the results screen shows why the factory ends up drowning in crates.

At the demo. Four classmates play retailer, wholesaler, distributor and factory for 20 rounds. The results screen then shows everyone's orders and inventory over time, and the bullwhip effect (small demand changes growing into big order swings upstream) appears on the chart.

The core.

- Four roles, a list of customer demand per round, a two-round shipping delay and a two-round order delay, with inventory and backlog (orders you could not fill yet) per role.
- Computer players that fill empty seats with a simple ordering rule, so one human can play alone.
- A results screen with the cost per role and the chart of order swings.

The stretch. A computer player that sees only its own orders and still beats the humans on cost; four laptops joining one game over the classroom wifi.

Work for three.

- **Rules and costs:** owns the rules and the costs, checks them against the original MIT Beer Game, and plays 10 full games to tune delays and demand so the bullwhip effect shows but the game stays winnable.

- **Game engine:** owns the round logic and the computer players, and writes a test (a small script that checks a function against an answer you computed by hand) for one hand-played round.
- **Screens:** owns the UI (the screens and buttons) and the results screen, and runs a trial with four classmates in week two to find what confuses players.

Data and tools. No external data; streamlit or nicegui (both turn a Python script into a web page), pandas, plotly (interactive charts). Check in week one whether your library can serve four browsers at once, or plan on hot-seat play (players take turns at one laptop).

Watch out. A game where nothing goes wrong because the delays are too short. Playtest early, with real people.

2. Dispatch

The food-delivery startup you have followed all term gets its dispatch screen: orders pop up on a live map of Hamburg, couriers move, and you decide who takes what.

At the demo. A classmate dispatches for three minutes on the projector while the on-time rate and the kilometers tick up. Then your own dispatch rule plays the same order stream, and the two scores stand side by side.

The core.

- A map of Hamburg with couriers moving along streets at a realistic speed, and orders appearing from a scripted stream, each with a promised delivery time.
- Click-to-assign dispatching with a live score: on-time rate and kilometers driven.
- The same order stream replayable with a fixed random seed (the same random sequence on every run), so human and algorithm face identical shifts.

The stretch. A dispatch rule of your own (nearest free courier, then something smarter) that beats the human; a rush-hour mode where rain slows the couriers.

Work for three.

- **Streets and speed:** owns the map and the courier movement, and times real cycling on five Hamburg routes with a phone to calibrate the speed.
- **Rules and balancing:** owns the order stream, the scoring and the clock, and runs 15 playtests so a good dispatcher reaches about 80 percent on time, not 100 or 20.
- **Algorithm:** owns the dispatch rule and the replay, and shows on 50 computer-played shifts that the smart rule beats nearest-free.

Data and tools. Real road routes from OSRM (free road routing on OpenStreetMap data): one request per second, and no key needed (the personal code some web services ask for). Cache the routes (save them to a file so they are never fetched twice), or use a simplified street grid. pygame (a library for 2D games) or a browser map with folium (maps in the browser).

Watch out. Couriers that fly in straight lines make every score meaningless. Fetch real routes once and cache them.

3. Logistics Exchange

Paper trading (play money at real prices) in four logistics stocks and oil, with a game master who drops news cards so you trade on a story and not on noise.

At the demo. Six classmates hold portfolios. The game master plays “port strike in Rotterdam” and “Suez closed”, the prices replay what really happened in such weeks, and the leaderboard reorders live.

The core.

- Accounts, buy and sell orders, a portfolio page and a leaderboard, priced from daily data via yfinance (free stock prices from Yahoo Finance).
- A news-card system: each card jumps to a chosen date range in the real price history, so every story has real prices behind it.
- A game master screen to play cards and freeze the round.

The stretch. A trading bot with a simple rule (buy the dip, sell on a spike) that competes with the humans; a chart of who panicked.

Work for three.

- **Markets and news:** owns the five prices and the card deck, and picks real dates for each card (what happened to Maersk and the Brent oil price around the 2021 Suez blockage), with sources.
- **Exchange engine:** owns accounts, orders and the portfolio math, with a test against a hand-computed trade sequence.
- **Floor and bots:** owns the UI, the leaderboard and the bot, and runs a five-minute test round with classmates in week two.

Data and tools. The yfinance tickers (the short codes of the stocks) are `MAERSK-B.CO` (Maersk), `HLAG.DE` (Hapag-Lloyd), `KNIN.SW` (Kuehne + Nagel), `DHL.DE` (DHL Group) and `BZ=F` (Brent crude oil). Download the history once in week one and cache it, because the demo must not depend on the wifi. streamlit, pandas, plotly.

Watch out. Trading on random daily noise teaches nothing. News cards tied to real dates are what make it a game.

4. Stack the Yard

Tetris for a container terminal: boxes arrive by ship and leave by truck, and every box you buried costs a crane move to dig out.

At the demo. A classmate works a three-minute shift on the projector. The end screen shows the reshuffles (crane moves that dig out a buried box) per box and the truck waiting time, next to the computer stacker’s result on the same arrivals.

The core.

- A yard of 6 rows with 5 stacks each, 4 boxes high; containers with a known pick-up time; ships that dump 20 boxes and trucks that fetch one each.
- A crane cost rule: every move costs time, digging costs extra, waiting trucks cost money.

- A score screen, and a fixed random seed so human and computer face the same shift.

The stretch. A computer stacker that sorts by departure time, for you to beat; a replay of your shift at ten times the speed.

Work for three.

- **Rules and balancing:** owns the arrival rates and the scoring, plays 15 shifts and tunes them so a good player gets under 1.5 reshuffles per box.
- **Computer stacker:** owns the stacking rule and the replay, and shows on 100 computer-played shifts that it beats random placement.
- **Yard screen:** owns the drawing in pygame, the controls and the score screen, with a test for the dig-out cost function.

Data and tools. No external data; pygame, numpy, random.

Watch out. A pretty yard with no time pressure is a spreadsheet, not a game. Give the shift a clock and make waiting trucks cost money.

5. Reverse the Trailer

Back a tractor and semitrailer into a loading dock, seen from above, with real trailer physics: steer left and the trailer swings right, overdo it and it jackknifes (folds against the tractor).

At the demo. A classmate plays three levels on the projector, from an open yard to a dock between two parked trailers. The scoreboard shows the time, the direction changes, the position and angle error, and the collisions.

The core.

- A kinematic model (it tracks positions, angles and speeds and ignores forces). The tractor is a bicycle model: one steered front wheel and one rear wheel. The trailer hangs on the kingpin, the pivot where it couples to the tractor.
- Each step the tractor moves by its speed and steering angle, and the trailer angle follows from that motion. A jackknife ends the attempt.
- Four levels (straight back in an open yard, a dock one lane to the side, a dock around a 90-degree corner, a dock between two parked trailers), each in three difficulties: easy with a guide line and a wide dock, normal, and hard with a time limit and a tight tolerance.
- A score per level and a high-score table.

The stretch. An autopilot that parks by itself and races the human; a drawbar trailer, which has two pivots and is much harder to back.

Work for three.

- **Vehicle and physics:** sources the real dimensions and steering limits, and checks the model on known cases: a straight reverse keeps the trailer angle at zero, and with a constant steering angle the truck settles on a circle that must match the turning circle on the data sheet.

- **Levels and balancing:** draws the yards to scale and tunes the tolerances from logged playtests (who clears which level in how many tries) until easy takes two tries and hard about ten.
- **Game and UI:** owns the pygame screen, the controls, the scoring and the replay of the best run, with tests of the collision check.

Data and tools. Vehicle dimensions from the EU rules (a trailer of 13.6 m, at most 16.5 m for tractor and trailer together) and from manufacturer data sheets; take the steering limit from the tractor's turning circle. pygame, numpy.

Watch out. A trailer that pivots around the wrong point feels wrong to anyone who has watched a truck back up. Run the straight-reverse and constant-steering checks before you design a single level, and keep them as tests.

Plan, load and route

These three turn an everyday operations decision (what goes where, in which order, at what time) into a tool that computes an answer and shows it. The agent can write the code that computes the answer; only you can check it against a case you worked out by hand.

6. Load the Trailer

Drag 33 Euro pallets onto a 13.6 m trailer without overloading an axle, then watch the computer do it better.

At the demo. A classmate loads a mixed order on the projector, and the axle gauge turns red on overload. The solver (a program that searches for the best arrangement) loads the same order in a second, and both loads are compared on weight used and legality.

The core.

- A 2D trailer floor (13.6 by 2.45 m) with pallet footprints (Euro 1.2 by 0.8, industrial 1.2 by 1.0), weights, and whether a pallet may be stacked.
- A calculator for the axle loads (the weight resting on each axle) from the pallet positions, checked against the German limits (40 t in total, 11.5 t on the drive axle).
- A greedy loader (takes the best spot now, never looks back) or OR-Tools (Google's free optimization library) as the solver, and a score.

The stretch. Delivery order (the last stop is loaded first); a height view with stacking.

Work for three.

- **Physics and law:** owns the axle-load formula, verifies it against two worked examples from a published loading guide, and turns them into tests.
- **Loading screen:** owns the drag-and-drop interface in pygame and the live gauge.
- **Solver and comparison:** owns the solver and runs 50 random orders to report success rate and weight used against the human.

Data and tools. Pallet and vehicle dimensions from public standards, no dataset; pygame, numpy, optionally OR-Tools.

Watch out. A wrong axle-load formula that the agent wrote with full confidence. The two worked examples are your only protection.

7. Monday Route Sheet

A real Hamburg florist, bakery or Tafel (food bank) drops its delivery list as Excel and gets back a map, an ordered stop list and a printable driver sheet in thirty seconds.

At the demo. You load the real, anonymized list from the business you interviewed, show the route on a map, export the driver sheet, then add a stop live and re-plan.

The core.

- Excel in (address, time window, notes) via openpyxl (reads and writes Excel files) and pandas.
- Geocoding (turning an address into coordinates) with a cache, and a stop order from a TSP solver (finds the shortest round trip through all stops) on real road distances.
- A folium map and a one-page driver sheet.

The stretch. Two vans with time windows via OR-Tools; a weekly log of kilometers per stop.

Work for three.

- **Customer:** finds a real business in week one, brings back its real list, runs a user test in week three and writes down what broke.
- **Pipeline:** owns geocoding and routing, with a cache so each address is looked up once, and a test that 20 known Hamburg addresses land within 100 m.
- **Sheet and handover:** owns the UI, the driver sheet, the README and the handover to the business.

Data and tools. Nominatim (the free OpenStreetMap address lookup): one request per second, and each request must carry a User-Agent string (a short text that names your app). OSRM for road distances; OR-Tools, streamlit.

Watch out. No real business by the end of week one means a tool for an imagined user. Agree on a fallback customer on day one.

8. Charge When Cheap

Ten electric trucks must be full by 6 am: forecast tomorrow's hourly power price from today's wind forecast and charge in the cheapest hours.

At the demo. Fourteen days of real electricity prices, your forecast for tomorrow against the price actually published that afternoon, and the charging plan as a Gantt chart (a timeline of who charges when), with the euros saved against "plug in at 18:00".

The core.

- A downloader for two free sources: day-ahead prices (tomorrow's hourly electricity prices, published each afternoon) from SMARD, the Bundesnetzagentur's market data site, and wind and temperature forecasts from Open-Meteo.

- A forecast that starts with “same hour yesterday” as the baseline (the simplest guess, which you have to beat), then a tree-based model (built from many yes/no splits), scored on a held-out month (data the model never saw).
- A charging scheduler that fills the cheapest predicted hours under a grid limit (the most power the depot can draw at once), and the saving.

The stretch. Run it every day for two weeks and report the real saving; a battery that sells power back at the peak.

Work for three.

- **Data pipeline:** owns the downloads and checks by hand that time zones and 15-minute versus hourly values line up, because one hour off ruins everything.
- **Forecast:** owns the model and reports honestly whether it beats the baseline.
- **Scheduler:** owns the plan, the dashboard and the cost assumptions (battery size, grid limit), each with a source.

Data and tools. SMARD offers a free download and an API (a web address that returns data instead of a page), and so does Open-Meteo; neither needs a key. pandas, scikit-learn (the standard Python library for machine learning), plotly, streamlit.

Watch out. A time-zone slip that lets the model peek at tomorrow’s actual prices, so the forecast looks magical. The hand check catches it.

Simulate a real system

These two build a model of something real (a lunch queue, a shipping network) and are judged on how well the model matches reality. The numbers you collect or source yourselves, and the honesty of your comparison, matter more than the code.

9. Digital twin

An animated copy of a real place you can observe (the cafeteria at lunch, a supermarket checkout, a warehouse dock), fed with numbers you counted yourselves, with sliders for staff.

At the demo. The animation runs a lunch hour in one minute: people arrive, queue and get served. A classmate drags the staff slider from three to two and watches the queue explode, while a side panel shows waiting time and cost.

The core.

- A counting sheet and at least three real observation sessions: arrivals per minute, service time per person, number of staff serving.
- A simulation (a program that plays out the system step by step) with random arrivals and service times drawn from your counts, animated in pygame or plotly.
- Sliders for staff and opening hours, and a chart of the average and the worst waiting time.

The stretch. A staffing recommendation per half hour with its cost; a comparison of the model’s waiting times against a fourth session you counted but did not use to build it.

Work for three.

- **Field data:** designs the counting sheet, runs the sessions with a stopwatch, and documents what was hard to count.
- **Model:** owns the simulation and checks it on a case with a known answer (one person serving, steady arrivals) before trusting it.
- **Animation and sliders:** owns the screen and the recommendation panel, and asks the cafeteria staff whether the numbers look right.

Data and tools. Your own counts; numpy, pandas, pygame or plotly, streamlit.

Watch out. A simulation that matches nothing because the counts came from one quiet Tuesday. Count three sessions and hold a fourth back.

10. Chokepoint

Close any passage you like (Suez, Hormuz, Malacca, Panama, one or several at once) and watch Hamburg's shipping lanes (its regular connections to other ports) reroute, lengthen or lose their route altogether.

At the demo. A world map shows Hamburg's 25 lanes along real sea routes. The class names passages to close, you tick them, and the routes redraw. A panel shows per lane the extra days, ships and fuel, or "stranded" when no sea route is left: closing Hormuz cuts the Gulf ports off instead of giving them a longer trip.

The core.

- A lane table you build yourselves: Hamburg to 25 ports with the weekly volume in TEU (twenty-foot container units), ship size and speed.
- A routing engine on searoute (a library that draws sea routes between ports and can avoid named passages): the route per lane with every passage open, then the route with your closures, and a label per lane: unaffected, rerouted with extra days, or stranded.
- The map and the panel, with several closures at once.

The stretch. A land bridge for stranded ports (Gulf cargo trucked to Fujairah, outside Hormuz, at a cost per container that you source); a view of the whole network in networkx (a library for networks of nodes and links) that finds the next weakest passage.

Work for three.

- **Lane table:** sources every number: volumes by region from Port of Hamburg press releases, ship sizes, speeds and fuel per day, each with its source and year.
- **Routing engine:** tests in week one what searoute returns when a passage has no alternative (version 1.6 returns a route of length 0 and no error, so a length of 0 means stranded), and checks three distances with every passage open against a published distance table.
- **Map and panel:** owns the scenario screen and the plain-words explanation of stranded versus delayed.

Data and tools. searoute can avoid Suez, Bab el-Mandeb, Hormuz, Malacca, Sunda, Panama, Bosporus and Gibraltar, but not the Kiel Canal or Dover, and it ignores a name it

does not know without an error. If you need a passage it lacks, build your own network of sea legs in networkx and remove a link. Port coordinates come from the World Port Index (a free table of the world's ports, downloaded once by hand); plotly, pandas.

Watch out. Big, confident fuel numbers that nobody can source. The README must separate sourced inputs from assumptions, lane by lane.

Machines that see

These four point a camera at the physical world: a container, a shelf, a parcel, a shipping label. The photos you take yourselves, and what you record by hand as the right answer, decide how good the result is.

11. Who Owns This Box

Photograph containers from the Elbe ferry and teach a model to tell Hapag-Lloyd from Maersk from MSC, then read the box number and tell whether it is a valid one.

At the demo. A classmate uploads a phone photo of a container. The app names the shipping line with a confidence bar, then reads the eleven-character number and says valid or invalid.

The core.

- A photo set you took yourselves, with a target of 60 photos per line for four lines, and a labeling sheet.
- A classifier (a model that sorts inputs into categories) in scikit-learn on resized photos, using color and shape features (the numbers a model takes as input).
- An upload page that shows the prediction, and the most common mistakes as a confusion matrix (a table of right and wrong guesses per class).

The stretch. easyocr (reads text from images) extracts the container number, and your own function checks its ISO 6346 check digit (the last digit, computed from the other ten); a view of which parts of the image drove the guess.

Work for three.

- **Photo campaign:** plans where, when and in what light to shoot so it is not 60 photos of one box, and labels them with a second person checking.
- **Model:** owns the features, the training (the model learning from your labeled photos) and an honest evaluation that trains on one shooting day and tests on another.
- **App and check digit:** owns the upload page, the check-digit code with tests on five known container numbers, and the text-reading step.

Data and tools. Your own photos; scikit-learn, Pillow and scikit-image (image handling); easyocr, which installs PyTorch (a large machine-learning library of about 2 GB, so install it on campus wifi before demo day).

Watch out. Photos from one afternoon under one sky give 95 percent in your test and 40 percent on the demo photo. Shoot on several days and test across days.

12. Shelf Gap

Photograph the same supermarket shelves every day from a taped mark and teach a model to see where the shelf is empty: the availability check a store manager does by eye.

At the demo. A fresh phone photo goes in. The app draws the grid over the shelf, marks each slot (one product's place on the shelf) full, low or empty, and shows how that shelf's availability moved over your ten days.

The core.

- A shooting plan: three to five shelf sections, a taped floor mark, the same height and zoom, twice a day for ten days, with the manager's permission.
- A hand-drawn grid per shelf that cuts every photo into slots, and a classifier on color and texture features with the labels full, low and empty.
- An upload page with the grid drawn over the photo and an availability curve (the share of filled slots per day).

The stretch. A chart of which products run out before the evening delivery; a second store to test whether the model also works there.

Work for three.

- **Shooting and labeling:** owns the plan and the labels, has a second person label 50 photos independently, and reports where the two disagree.
- **Grid and model:** owns the grid, the cropping and the training, and tests on days eight to ten after training on days one to seven.
- **Store view:** owns the UI, the grid over the photo and the curve, and asks the store staff whether the result matches what they know.

Data and tools. Your own photos; scikit-learn, scikit-image, Pillow, streamlit.

Watch out. A phone position that is off by a few centimeters shifts the grid, and the model learns shelf edges instead of products. Check the grid on every photo the same day, and reshoot at once when it is off.

13. Measure the Parcel

Photograph a parcel next to a printed marker of known size and get its length, width and height, then the carrier's size class and price, without a tape measure.

At the demo. A classmate puts any box on the table, you take one phone photo, and the screen shows the three dimensions, the DHL size class and the price. A tape measure settles whether you were right.

The core.

- Printed ArUco markers (black-and-white squares that the image library OpenCV finds and measures by itself) on the table and on top of the box, so the photo has a known scale.
- A measuring step: find the box edges next to the markers, convert pixels to centimeters, and output length, width and height with an error estimate.

- A lookup from the dimensions, and a weight you type in, to the carrier's public size classes and prices.

The stretch. A second photo from the side to fix the height; a live webcam mode that measures a box the moment it is put down.

Work for three.

- **True sizes:** tape-measures 40 boxes of different sizes and colors and keeps the sheet: these are the numbers your tool is judged against.
- **Measuring:** owns the marker detection and the pixel-to-centimeter math, and charts the error against camera angle and distance, so you know when a photo is good enough.
- **Price lookup and app:** owns the size-class table with its source and date, the UI, and the demo setup (where the box stands, where the phone stands).

Data and tools. Your own photos; opencv-python (the Python package of OpenCV, with the marker functions included from version 4.7), numpy, streamlit; the DHL size classes and prices from the public price list.

Watch out. Height from a single photo taken from above is the weak dimension, and the error grows fast at steep angles. Measure that error, show it in the demo, and plan the second photo or the marker on the box from week one.

14. Scan to Track

Hold a real DHL or Hermes label from your own parcel to the webcam: the app reads the tracking number, tells the carrier from the number, and adds the parcel to your own tracking dashboard with its live status.

At the demo. Three labels pass the webcam, and each is read, checked and added in seconds. The dashboard shows every parcel the group tracked over the three weeks with carrier, last status and days in transit, and one classmate's own parcel is added live.

The core.

- A webcam reader with zxing-cpp (a barcode library) that finds the barcodes on a label and reports each one's type (the bar pattern standard, such as GS1-128 or Interleaved 2 of 5) and content.
- Carrier recognition from the number itself: DHL numbers have 12 or 20 digits, Hermes numbers have 14. The 20-digit and the 14-digit numbers end in a check digit computed with the weights 3 and 1, so the app rejects a misread before it saves anything.
- A dashboard that stores only the tracking number, the carrier and the times, and pulls the live DHL status through the official DHL Shipment Tracking API (free key from developer.dhl.com, 250 requests a day, one every five seconds).

The stretch. The Hermes status through 17TRACK (a multi-carrier tracking service with a small free quota for new accounts); a chart of transit days per carrier from your own parcels.

Work for three.

- **Labels and privacy:** collects 40 labels from the group's own parcels or with the owner's consent, writes the rule that only the number and the carrier are stored and the photo is never saved, and tests reading at angles and in bad light.
- **Numbers:** owns the carrier rules and the check-digit code with tests on 20 real numbers, and scans real labels in week one to see which barcode types the two carriers print, because Hermes does not document its barcode publicly.
- **Dashboard and APIs:** registers the DHL key in week one (DHL asks for a company name in the profile, so enter the university and use your university e-mail), builds the dashboard and caches every status request.

Data and tools. Your own labels; zxing-cpp (reads Code 128 and GS1-128, Interleaved 2 of 5, DataMatrix and QR), opencv-python for the webcam, streamlit, pandas, a CSV file as the store. For Hermes without a key, store the number and link to the Hermes tracking page; do the same for DHL if the key does not come through.

Watch out. Labels show names and addresses, and a webcam frame is a photo of them. Never write the image to disk, show the group's privacy rule on your first slide, and use only labels from your own parcels or with the owner's consent.

Bibliography