WASP Software Engineering Course Module 2025

Chen Ling
chenlin@kth.se

August 2025

1 Introduction

My research is aiming to build a practical pipeline to infer latent emotion from non-verbal behavior by coupling embodied 3D perception with deep generative analysis.

To develop this pipeline, the implementation will comprise two stages focused on emotion analysis from visual data. The first is a 2D-to-3D geometry extractor that reconstructs a consistent 3D representation of the face—and, when appropriate, body pose and hand gestures—from images or video, producing stable features such as expression/shape parameters and head pose. The second is an emotion recognition module that maps these 3D features to latent affect (e.g., valence–arousal or discrete categories) with temporal modeling; Facial Action Units can serve as intermediate supervision to heighten sensitivity to subtle movements. Training will jointly optimize a forward behavior model and an inverse inference path so the forward model regularizes the inverse mapping, improving stability and data efficiency.

This system will be evaluated and applied in three scenarios that share the same core. (1) A digital-agent setting where affect estimates condition a partner avatar during dialogue; here I pre-train on public interaction corpora and fine-tune on in-lab recordings. (2) A neuroscience study that infers stimulus valence from head motion and facial dynamics during approach–avoidance tasks. (3) A driver-frustration detector co-developed with an industry rig, initialized on neutral driving and fine-tuned on provoked frustration.

2 Lecture principles

Validation-centric QA is the practice of verifying the system truly solves the right problem under real-world variation—by measuring performance across meaningful data slices and by checking invariants (properties that should stay stable under small, controlled changes).

2.1 Validation-centric QA: slicing

In my work, one overall score does not tell the truth. Lab videos are clean and frontal; wild videos have motion, compression, and clutter. When I slice results by setting (lab vs. wild), the gap is obvious. Slicing by pose also helps: performance drops once yaw/pitch get large; many errors come from unstable 2D→3D fits. Occlusion/eyewear is another slice: reflective glasses and hand-to-face make AU recall worse. Image quality matters too: low resolution or heavy compression gives noisy micro-movements. Slicing turns one big score into a list of where things break. It shows where the pipeline actually breaks (environment, pose, occlusion, quality) and whether the break happens in the 2D→3D fit or in the emotion classifier

2.2 Validation-centric QA: invariants

Invariants make these issues concrete. A horizontal flip should swap left/right AUs but keep global valence roughly the same; if valence shifts, something is wrong in the mapping. Small rotate/scale tweaks on the same frame should not change expression parameters much; big drift points to cropping or normalization problems. Frame-rate changes (e.g., 60→30 fps) should keep the arousal curve shape; if it bends, the temporal modeling is too sensitive to step size. Mild lighting jitter should keep embeddings close; if not, photometric robustness is uneven. These checks explain failures better than a single held-out score because they expose representation quirks instead of averaging them away.

3 Guest-Lecture Principles

3.1 Problem Space vs. Solution Space

The guest talk splits what/why from how. That helps me keep the story straight. Context (why): a. Avatar cares about smooth turn-taking. b. Neuroscience cares about a clean, aligned emotion curve. c. Driving cares about few false alarms. Requirements (what): What do we output: VA values or a few states? What latency is OK? How stable should results be across pose/occlusion/compression? System (how): Only here come the choices: 2D→3D face/body features, AU as an intermediate signal, temporal modeling.

When a model looks great on lab clips but drops on wild videos, it is often a requirements miss (we never said "stable under large yaw and glasses"), not a model issue. Framing results this way makes reviews simpler: we can point to which problem statement was unmet instead of swapping architectures at random.

3.2 Goal Modeling and System Vision

The talk also pushes us to write goals and draw the system boundary. That turns vague wishes into concrete use cases. Stakeholders & goals. a. Avatar:

natural feel (usage), low delay + steady behavior (system). b. Neuroscience: time-aligned curves (usage/system), privacy (system/business). c. Driving: low false alarms, traceable decisions (system/business). Conflicts. Low latency vs. heavy smoothing. Detailed logs vs. privacy. Naming the trade-offs early avoids surprises later. System vision. Inside: 2D→3D features, emotion recognizer, slice reports. Outside: data capture, dialogue manager, vehicle bus. Simple use cases keep it concrete: "inspect per-slice performance," "check dialogue latency," "export lab vs. wild report."

This framing fits day-to-day work: when results slip under large yaw or with reflective glasses, I can say which goal is failing and whether it sits inside the system boundary. It also lines up with my evaluation style (slicing and invariants): those checks are part of the what we promise, not an afterthought.

4 Data Scientists versus Software Engineers

Do you agree on the essential differences between data scientists and software engineers? Why or why not?
Mostly yes, but with caveats. The chapters describe a real split in focus: data scientists optimize models and evidence (metrics, experiments, papers), while software engineers optimize systems and delivery (requirements, reliability, latency, cost). That lens is useful; it explains why "great accuracy" can still fail in production. But in practice the boundary is blurry. Hiring markets often ask for ML engineers who can do both: solid SE fundamentals plus model work. I take the distinction as a helpful abstraction, not a hard line. Incentives, artifacts, and time horizons differ—but many roles sit in the middle.

Will these roles specialise further or merge? Explain.
Both, depending on context. Large or regulated organisations will keep strong specialisation; small/startup teams will blend roles and rely on tight collaboration. Overall, the overlap keeps growing while each side still has a deep track—i.e., more *T-shaped* people with shared basics and one area of depth.

5 Paper Analysis

5.1 2024 CAIN An Exploratory Study of Dataset and Model Management in OSS ML Apps[1]

1. Core ideas and their SE importance
The paper examines how open-source ML apps store and track datasets and models. The authors read GitHub repos and noted where files live (in-repo or external), whether they're under a version control system (VCS), and how often they change. They found scattered storage, large files pushed to remote drives, few artifacts tied to VCS history, rare updates, heavy use of pre-trained models, and frequent link rot—making it hard to reproduce results, trace which data/model were used, keep assets available, or pass audits.

SE importance: without basic care for data/model artifacts (versions, checksums, a stable place), you can't reliably rerun results, trace what was used, keep CI stable, or hand work over—so this "boring" management is essential.

2. Relation to my research

My pipeline uses the same artifacts as in the paper: datasets (lab and in-the-wild video) and models (2D→3D extractors, AU encoders, emotion recognizers, plus pre-trained backbones). I run into the same problems: files live in ad-hoc places, versions are unclear, links break, and I depend on many pre-trained models (models trained by others). When links move or names are reused, I can't tell which video subset, preprocessing script, or weight file produced a result. That kills reproducibility and makes slice checks (by pose, occlusion, lighting) unreliable. The files are large, so flaky remote storage also breaks CI (automated builds/tests) and slows onboarding.

This matches my hands-on experience. I've had bash scripts that auto-download assets fail because a wget link went dead or redirected. I've also seen weight files with the same name but different contents, and "latest" folders that changed silently. Pre-trained weights scattered across Drive/Dropbox/HF made it hard to pin what I actually used.

What would help is simple and concrete: keep data and models in one place (registry or VCS with large-file support), add a version tag and checksum for each file, record minimal metadata (source, license, schema), and stop relying on "latest" links.

3. Integration into a larger AI-intensive project

A larger AI-intensive project here is the same as before: an affect-aware conversational agent (with a driver-monitoring variant). It takes video, builds 2D→3D face/body geometry, and outputs valence/arousal to drive dialogue or in-cabin alerts. To make this workable, I keep datasets and models in one place (a registry or Git with large-file support), each with a fixed version, a checksum, and a few lines of metadata (source, license, schema, slice tags). Before any training or release, a simple CI job checks that links still work, hashes match, and schemas are consistent. My WASP piece is the emotion service in this setup: it trains and evaluates against those exact artifact versions, and it produces slice reports (pose/occlusion/illumination) so robustness is visible at the system level, not just as one average score.

4. Adaptation of my research

This paper's message is simple enough for me to use: treat data and models like part of the build, not like loose files on the side. If I do that, the work gets easier to repeat, compare (lab vs. wild), and share with other teams.

To make this concrete, I can start small: I'll put datasets and weight files in one stable place instead of scattering links. No more "latest" folders—each file gets a version in the name and a checksum next to it. For pre-trained models, I'll pin the exact source and version and keep a mirrored copy, so a dead Drive link doesn't stop the pipeline. Alongside each artifact I'll keep a tiny "card": source, license, a note on schema, and the slice tags I care about (pose, occlusion,

illumination). This is just a few lines in a `README` or a `metadata.json`, nothing fancy.

Before any training or release, a small CI step can do the basics: the link still resolves, the hash matches, the schema looks right. If it fails, we fix the asset first, not after a half-day run. Each experiment will write a short run manifest (commit, data/model versions, seed, key hyperparams). Future-me can rerun it without guessing which video subset or which weight file I used. I'll add a short onboarding note and a "what's live" snippet in the repo that lists the current data/model versions the project uses, so new people don't have to dig.

For day-to-day work, clear names and folders help more than people think: `data/lab/v1/...`, `data/wild/v2/...`, `models/face3d_v0.3/...`. If I really need larger tooling later, I can layer on Git LFS or DVC; I don't have to start there. The important thing is that lab vs. wild comparisons point to explicit versions and that slice tags travel with the data. That way, when results diverge, I can say "this changed in strong backlight with glasses," not just "the metric dropped."

I'll also plan a gentle clean-up path: don't rewrite history, but for any new run, use the stable store and write the small metadata; backfill older assets when I touch them. Weekly, I can scan for dead links and mismatched hashes and file a small issue if something is off. Backups are simple too: one bucket, lifecycle rules, and read-only links for releases. None of this needs heavy process—it's just giving data and models a home, a name, and a receipt. That alone cuts most of the "can't rerun / link is dead / which weights?" friction and makes my results easier to share and trust.

5.2 2025 CAIN How Do Model Export Formats Impact the Development of ML-Enabled Systems? A Case Study on Model Integration[2]

1. Core ideas and their SE importance

The paper asks a practical question: how do *model export formats* change day-to-day integration work? The authors build two small apps (numeric predictor, sentiment), implement them in three stacks (Python/Flask, Node.js, Next.js), and try five export formats (ONNX, TensorFlow SavedModel, TorchScript, Pickle, Joblib)—30 trials in total. The pattern is consistent: **ONNX** is the easiest when you leave Python or target JS; **SavedModel** is smooth inside the TensorFlow world and can bundle some preprocessing; **TorchScript** fits PyTorch in Python/C++; **Pickle/Joblib** are Python-only and fragile elsewhere. A recurring issue is *pre/post-processing*: when it lives as ad-hoc Python code, portability drops; when it's captured in the exported graph, integration gets simpler.

Why this matters for software engineering is practical: *how a trained model is packaged for use outside the training code* determines how easily it moves across stacks, how much it costs to maintain, and how safely you can run it. Picking one or two standard export formats and writing down the expected

inputs/outputs cuts glue code, avoids ad-hoc subprocess workarounds, and lets the same model run across teams and platforms.

2. Relation to my research
My system runs in two places: Python (for training and evaluation) and, sometimes, a web/ non-Python stack for integration. This paper gives me a simple rule of thumb: in pure Python, export PyTorch models as TorchScript and TensorFlow models as SavedModel; when the model needs to run in JS/TS or another language, export ONNX. Keep any small pre/post-processing steps next to the export and put a version on them, so nobody needs custom Python scripts later.

3. Integration into a larger AI-intensive project
Continuing the same project from Paper 1—an affect-aware conversational agent (with a driver-monitoring variant)—this paper helps me decide where to draw the line between training code and serving code. In the Python parts (offline training and batch evaluation), I keep models in their native exports (TorchScript for PyTorch, SavedModel for TensorFlow). For the web-facing agent built in Node/Next, I ship ONNX so it runs without a Python sidecar.

Small preprocessing or postprocessing steps (resize, face alignment, normalization, simple AU re-mapping) should either be inside the export or shipped as a tiny, versioned helper, so teams don't need ad-hoc glue scripts.

Before a release, I do three quick checks: the export loads in its target stack; the I/O shapes and units match what we agreed; and a couple of sample inputs produce outputs with sane ranges. In this setup, my WASP piece is the emotion service: a clean API that takes frames and returns valence/arousal, plus a simple slice summary (by pose/occlusion/illumination) so product teams see where the model is strong and where it is fragile.

4. Adaptation of my research
This paper's practical message is to decide early how a trained model is packaged for deployment and to keep those choices small and consistent. In Python services I keep native exports (TorchScript for PyTorch, SavedModel for TensorFlow); when the model must run in JS/TypeScript or another non-Python part, I export ONNX. Where possible I bundle small preprocessing/postprocessing with the export so the I/O is clear and teams don't need ad-hoc glue. A tiny export script and a couple of quick checks (loads in target stack, I/O matches, outputs look sane) make the hand-off repeatable.

I'll also tame preprocessing. The small steps—resize, face alignment, normalization, simple AU re-mapping—shouldn't live as scattered scripts. Where possible I'll fold them into the export so the model really defines its own inputs and outputs. If something must stay outside, I'll ship it as a tiny, versioned helper rather than one-off glue. Alongside each export I'll keep a one-page "I/O card": input shape, color order, normalization, units (degrees vs. radians), coordinate frame (camera vs. head), and what the outputs mean. That card answers most integration questions without a meeting.

To make this routine, I'll add a small export script: give it a trained checkpoint, it produces the selected exports and a short metadata file (model

name/version, framework, export format, runtime requirements, license, and where the files live). Before handing anything over, I'll run two or three quick checks: does the export load in its target stack, do the inputs/outputs match the I/O card, and do a couple of sample inputs produce sane-looking outputs (shapes and scales make sense, no NaNs). If something is off, I fix it before anyone else touches it.

Finally, I'll smooth hand-offs: keep a minimal "how to load" example for Python/Flask, one for Node.js, and one for Next.js; note a simple fallback (if JS hits an unsupported operator, call a Python service instead of spawning ad-hoc subprocesses); avoid risky formats outside trusted Python (no Pickle/Joblib in web code); and jot down a small table with export size and basic load/latency numbers so I can compare over time. None of this is heavy process—it's just agreeing on one or two export paths, writing down the I/O, and running the same small checks every time. That's enough to keep my emotion pipeline portable and easy to plug into other stacks without extra glue.

6 Research Ethics & Synthesis Reflection

Search and screening. I used the CAIN 2024/2025 Accepted Papers pages (short/long clearly marked), skimmed titles, shortlisted general AI-engineering topics, read abstracts to verify scope and contribution, kept only full/long items, and then fetched PDFs from IEEE Xplore.

Pitfalls and adjustments. Because I am not fluent in some terms, I relied on abstracts and contribution statements. I also noticed a 2025 tilt toward LLM-focu sed full papers (an industry trend); since these are less relevant to my topic, I filtered LLM-heavy items in the first pass.

Ethical considerations. I used an LLM as a proofreading tool to improve grammar and clarity. The ideas, structure, and final wording are my own, and no passages were directly generated or copied from the LLM.

References

[1] T. R. Toma and C.-P. Bezemer, "An exploratory study of dataset and model management in open source machine learning applications," in *2024 IEEE/ACM 3rd International Conference on AI Engineering – Software Engineering for AI (CAIN)*, 2024, pp. 64–74.

[2] S. K. Parida, I. Gerostathopoulos, and J. Bogner, "How do model export formats impact the development of ml-enabled systems? a case study on model integration," in *2025 IEEE/ACM 4th International Conference on AI Engineering – Software Engineering for AI (CAIN)*, 2025, pp. 48–59. DOI: 10.1109/CAIN66642.2025.00014.