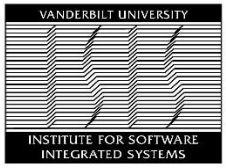


Towards the Co-evolution of Models, Code, and Assurance Cases: The CAID Framework

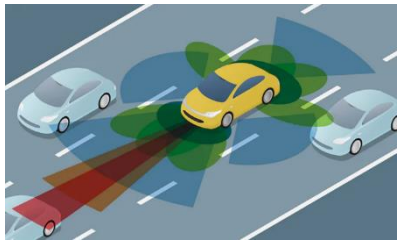
Patrik Meijer, Nag Mahadevan, Mark Wutka, Gabor Karsai
Institute for Software-Integrated Systems
Vanderbilt University
Supported by DARPA Assured Autonomy Program

Outline

- The challenge:
  - High-assurance System Software and CI/CD
- Paradigm for assured software
  - Artifacts: models + implementation + assurance arguments
- Challenges of CI/CD
  - Continuous evolution → Continuous assurance
  - Dynamic maintenance of assurance arguments
- CAID: Next-gen development – CI/CA/CD
  - Integration/coordination across tools
  - Example scenario
- Results
  - Assurance argument construction, editing, and review
  - Integrating development tools – with dependency tracking
- Conclusions

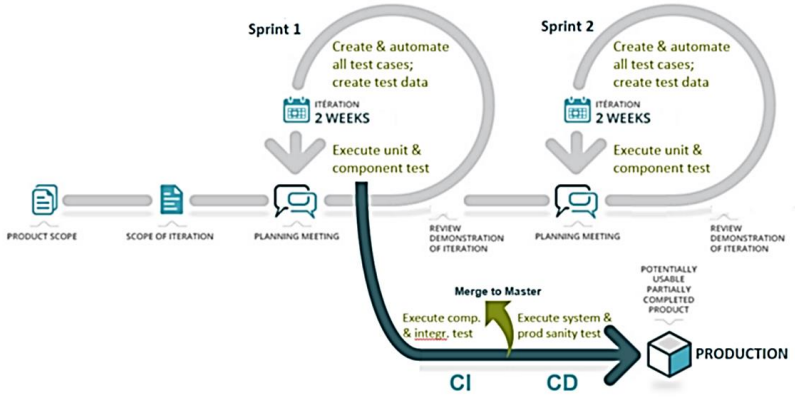
The challenge

- High-assurance Software Systems
  - Safety-/mission-critical systems where consequences of failures are catastrophic
  - Examples
    - Advanced Driver Assistance Systems (ADAS)
    - Cockpit automation systems
    - Power grid / protection systems
    - Healthcare CPS





- On the other hand…
  - Continuous Integration / Continuous Delivery
  - Agile development



System safety engineering today

- Often post-development
- Independent safety review
- Often mandated by government regulations
- Challenge: Software as a 'system integrator'

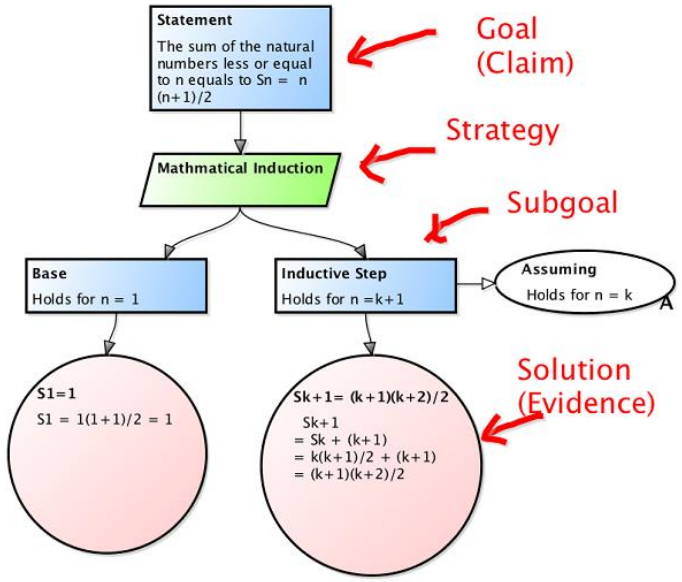
Goal Structuring Notation :

A graphical tool to represent a logical argument

4+1 types of nodes:

- **Goal**: What we want to prove ('safety claim')
- **Assumption/Context**: Under what circumstances
- **Strategy**: How we go about proving the goal
- **Solution**: Evidence to support a goal
- **Sub-goals**: decomposition of a higher level goal

Source: http://www.goalstructuringnotation.info/



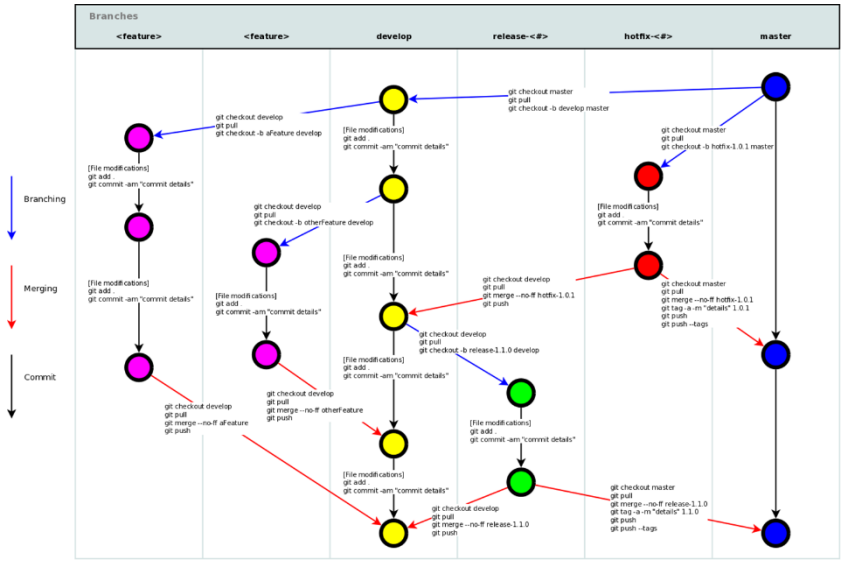
Assurance argument – 'Documentation' for HASS?

Engineering Artifacts needed for Assurance

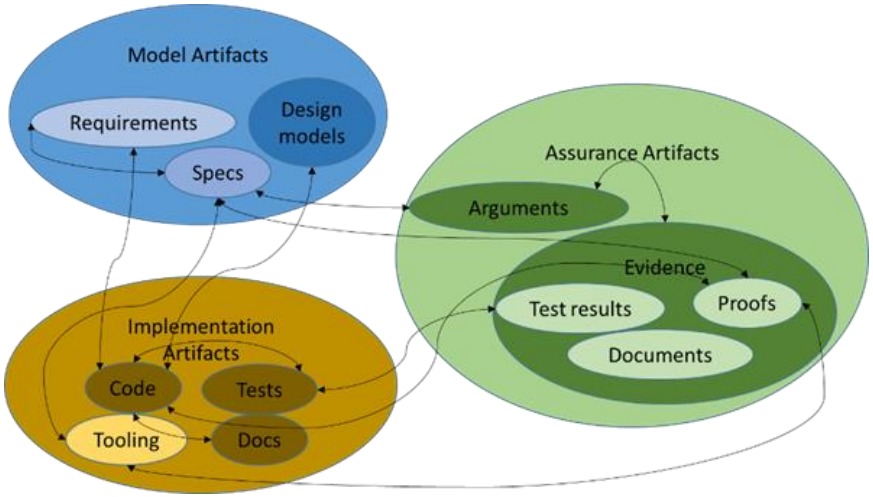
Artifact	Role
Model artifacts	
Requirements	Expectations: functions, performance, behavior, …
Specifications	Precise formulation of requirements
Design models	Representation of design decisions on architecture, functions, interfaces, …
Implementation artifacts	
Code	'Production code' … maybe generated
Tests	Unit/system-level tests to show lack of flaws
Tooling	Tools and their 'settings' used to build the system
Documentation	Code-level and end-user documentation
Assurance artifacts	
Assurance arguments	Claims and logical (possibly informal) arguments for their validity
Evidence	
Proofs	Formal logical arguments / models checked
Test results	Reproducible records of test runs
Documents	Other evidence sources (e.g. datasheets, etc.)

Observations

1. The artifacts are produced (and maintained) in a continuous development process
   - Version controlled, continuous development and integration

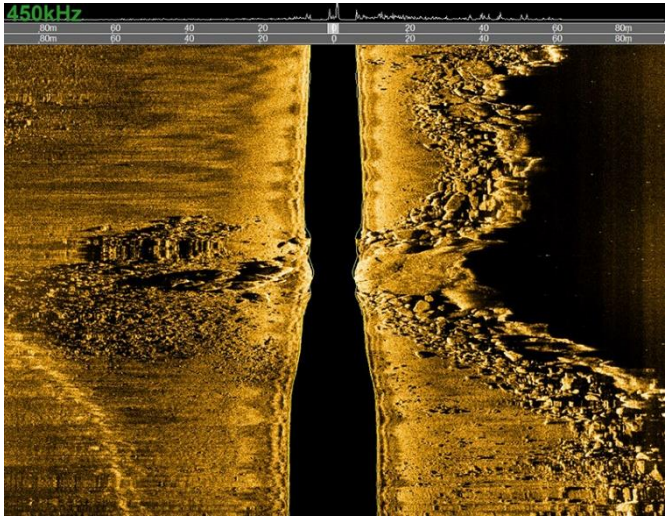


2. The artifacts are in complex dependency relationships
   - Explicit representation and management of these dependencies is inevitable



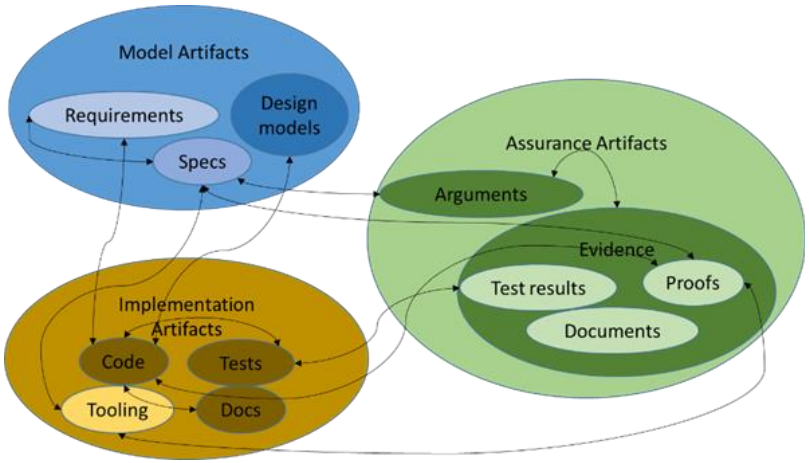
Example: Add a new 'mission type' for an Autonomous Underwater Vehicle (AUV)

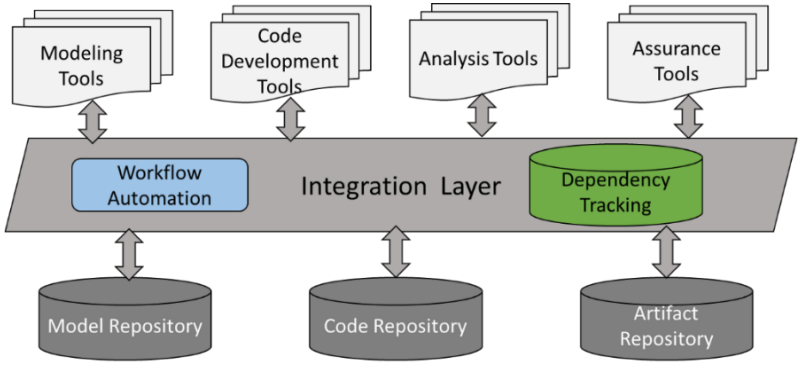
- Task: Underwater infrastructure (pipeline) inspection
- Requirements:
  - Descend close to sea floor
  - Find the infrastructure (cable, pipe, etc.) object
  - Inspect object, up to a distance, limited by battery charge
  - Monitor battery charge and control surfaces for degradation
  - Safely return home, under all scenarios
- Steps:
  - Add new sensor: Side-Scan Sonar
  - Spec: performance, safety, ... goals
  - Change software architecture
  - Integrate new sensor
  - Update autonomy logic
  - Devise new tests/verification regimes
  - Revise 'safety assurance arguments'
- Integrate all these into the CI/CD



Vision:
Continuous Assurance-Integrated Development

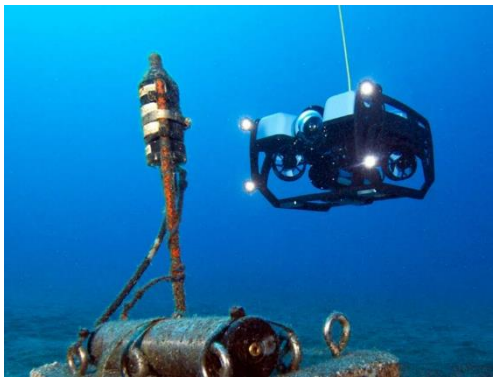
- Tools:
  - Modeling tools for capturing requirements, formalizing specifications, and representing designs in high-level models
  - Development tools for code and test construction (generation), static/dynamic code analysis, model and code verification, documentation production
  - Assurance tools for constructing, reviewing, and archiving assurance evidence data sets

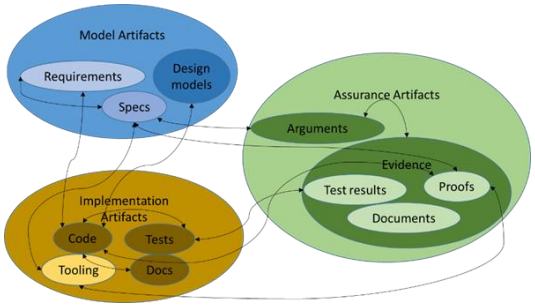




Notional use case: Autonomous Underwater Vehicle (AUV)

- Requirements:
  - Descend close to sea floor
  - Find the infrastructure (cable, pipe, etc.)
  - Inspect object, up to a distance, limited by battery charge
  - Monitor battery charge and control surfaces for degradation
  - Safely return home, under all scenarios
- Software functions for the 'Safe return':
  - Monitor battery health and compute remaining useful charge
  - Continuously estimate/plan safe return trajectory
  - Control vehicle movement and switch to 'return-to-home' mode, if needed
- Elements of an *assurance argument* for the 'Safe return' use case:
  - (1) correct estimation of remaining useful charge in the battery,
  - (2) correct calculation of the safe return trajectory,
  - (3) correct reaction of the vehicle controller to critical battery charge levels under all foreseeable modes of operation, and
  - (4) the correct integration of the above



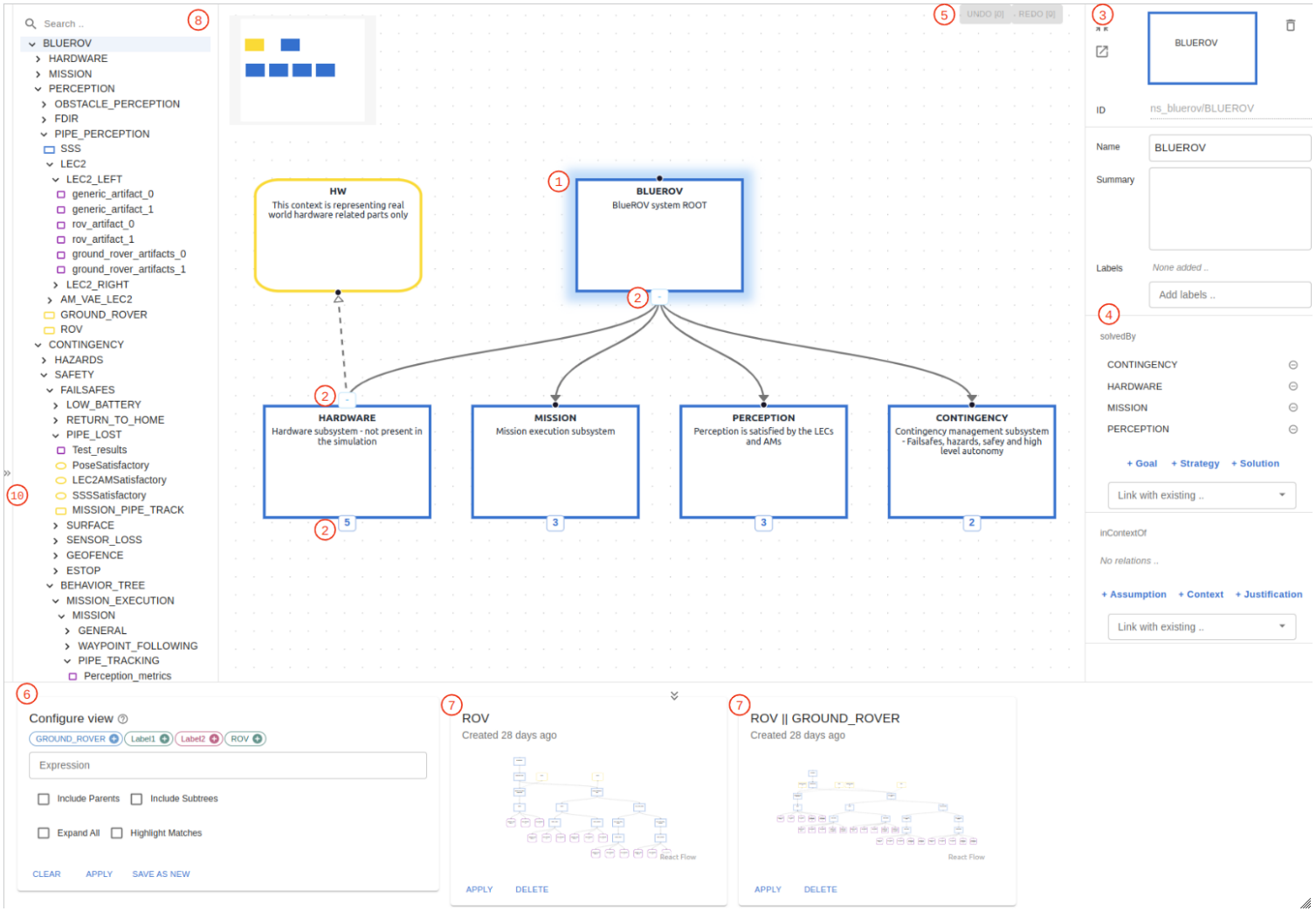


<u>Tool use case: Traceability</u>

Requirement → System function → Software model → Software component → Test case → Test result (evidence) → Supported assurance claim

- Tracking the impact of a change (forward propagation)
- Dependency analysis (backward propagation)
- History of changes (append-only log of versions/changes)

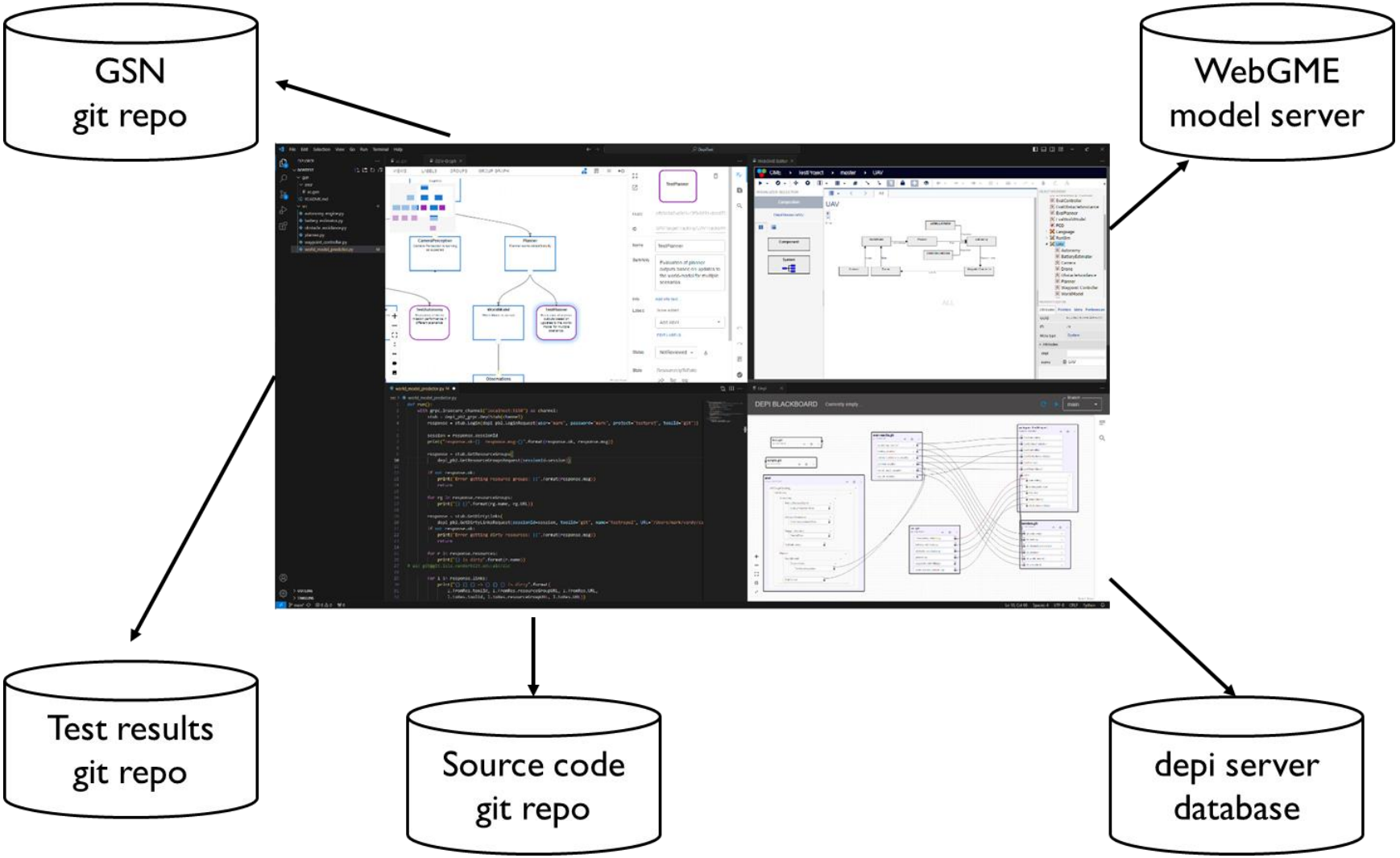
Implementation:
Assurance Case Construction Tool



Implementation: Assurance Case Construction Tool

- The central canvas showcases a tree-graph, typically a single-rooted tree with a top-level goal, like the BLUEROV in this example (1).
- Users can navigate through the tree using the expand/collapse buttons (2). The blue outline around a node indicates its selection, and its properties (3) and relationships (4) become editable in the right-hand panel. The action buttons in (3) let the user filter the model to display only a graph's subtree and quickly locate the line in the textual document where the node is defined.
- Any edits made in the graphical editor are instantly synchronized with the textual model and the associated .gsn files. The VS Code extension tracks these updates, adding them to an undo stack. This feature allows users to undo or redo their changes without needing to navigate through the textual .gsn files (5).
- Each node can be assigned a set of labels that can be referenced from a view (6). The view's core component is the expression, a logical operation (and, or, not) based on the labels defined in the model. Essentially, a view acts as a filter, displaying only nodes with labels that satisfy the specified expression (7). These views are saved with the model and can be reapplied to the main graph.
- The left panel displays a compact overview of the GSN model as a tree browser (8). Users can navigate this tree similarly to the main canvas, with node selection and editing available through sections (3) and (4). A search field at the top allows users to see an expanded, filtered view of the tree browser, displaying only matches and their parent nodes. By default, the search field filters by name, but other options can be selected.
- To edit the information/details (10), users can bring up a multi-line text editor (note: only the access point is shown here, not the actual editor).

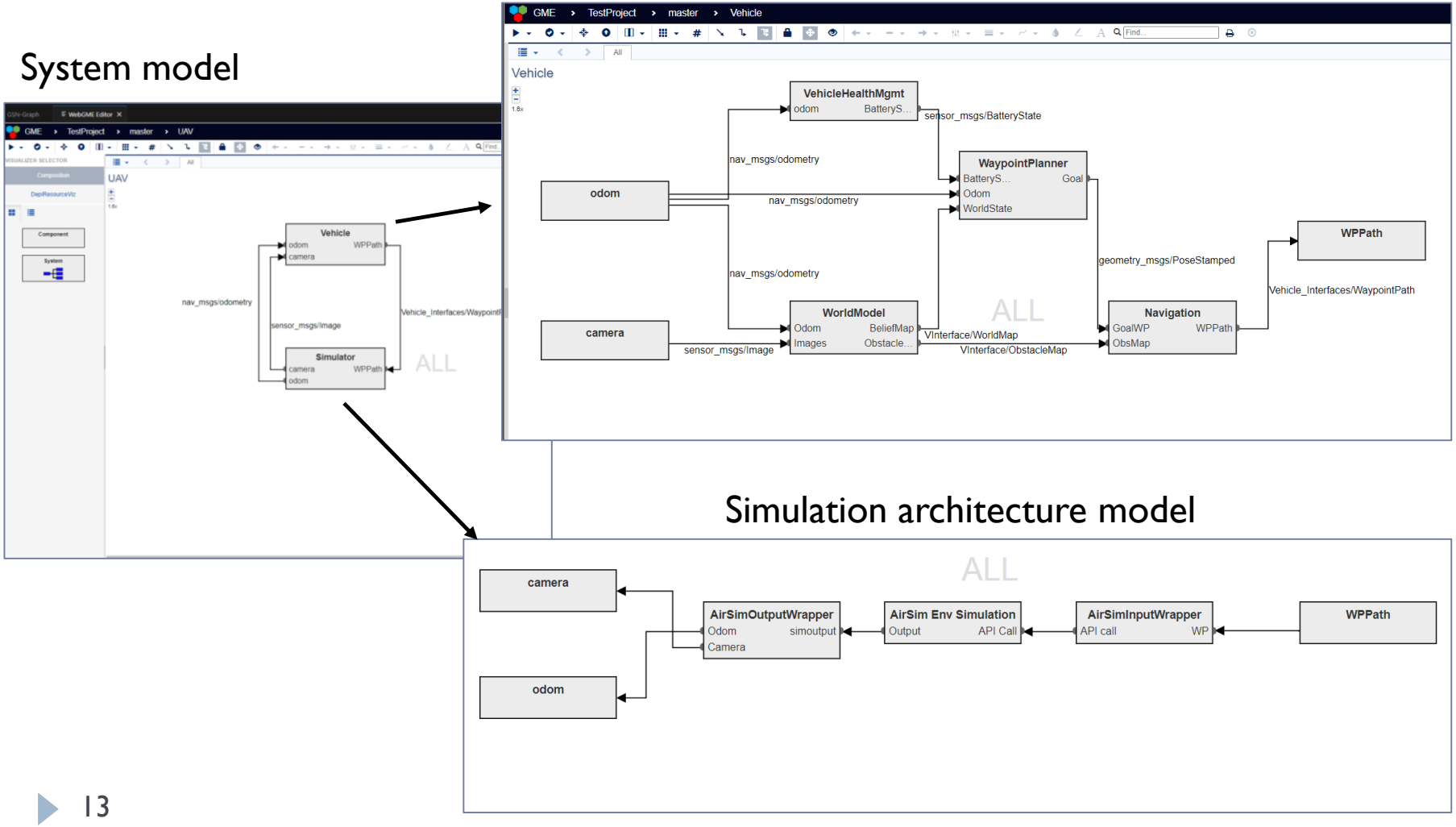
Implementation: Assurance Provenance Demonstration architecture



Assurance Provenance Demonstration : Architecture models

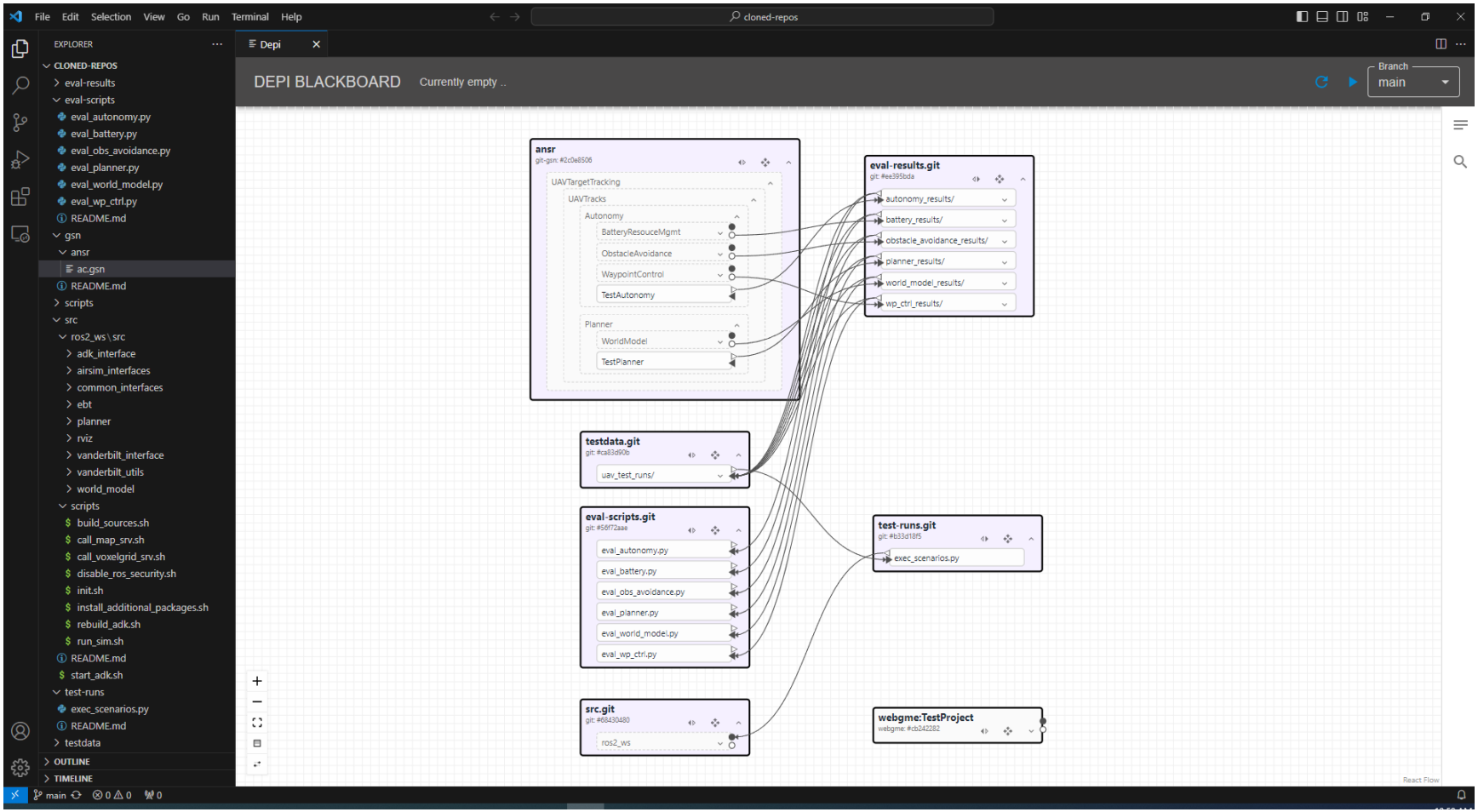
Vehicle software architecture model

System model

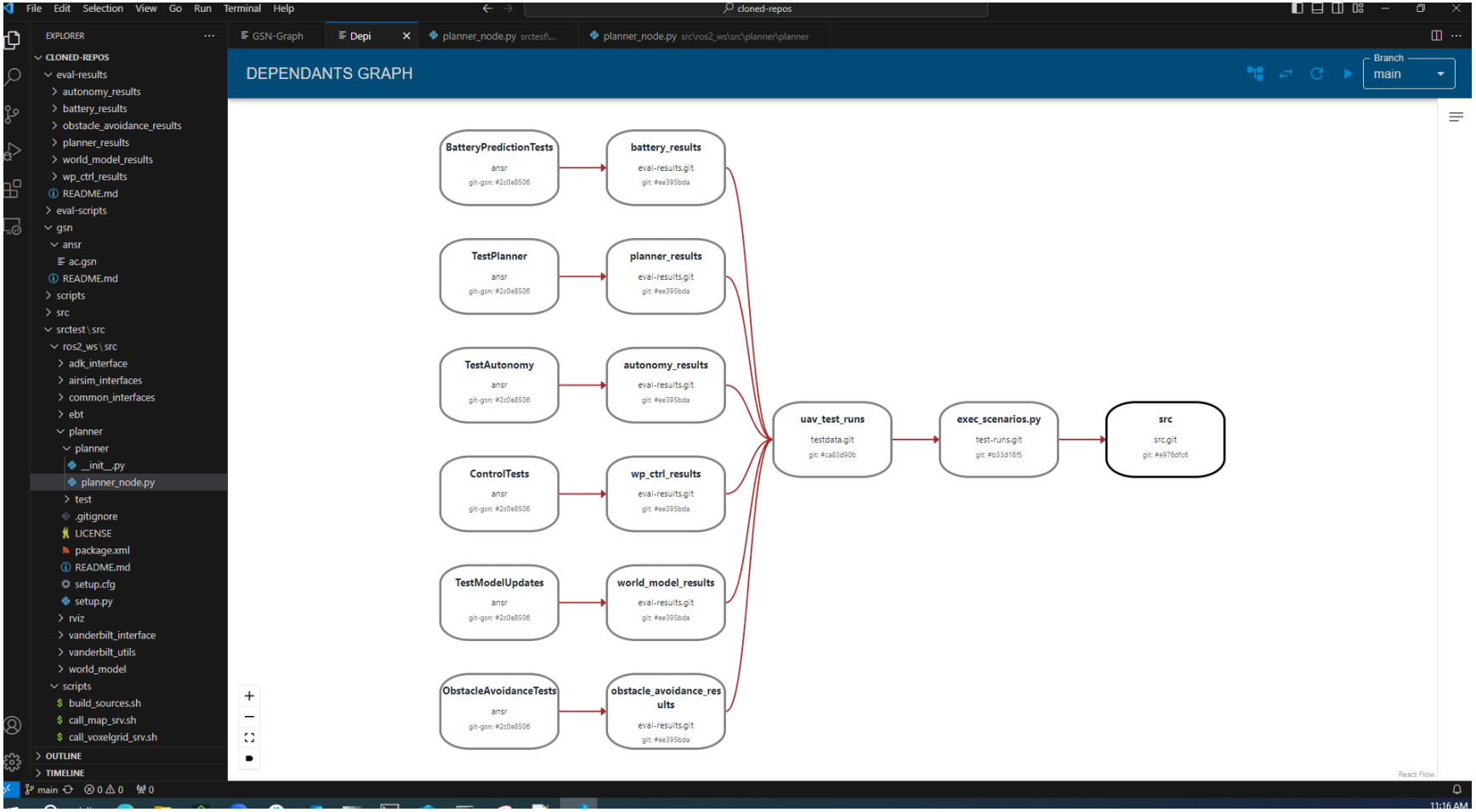


Simulation architecture model

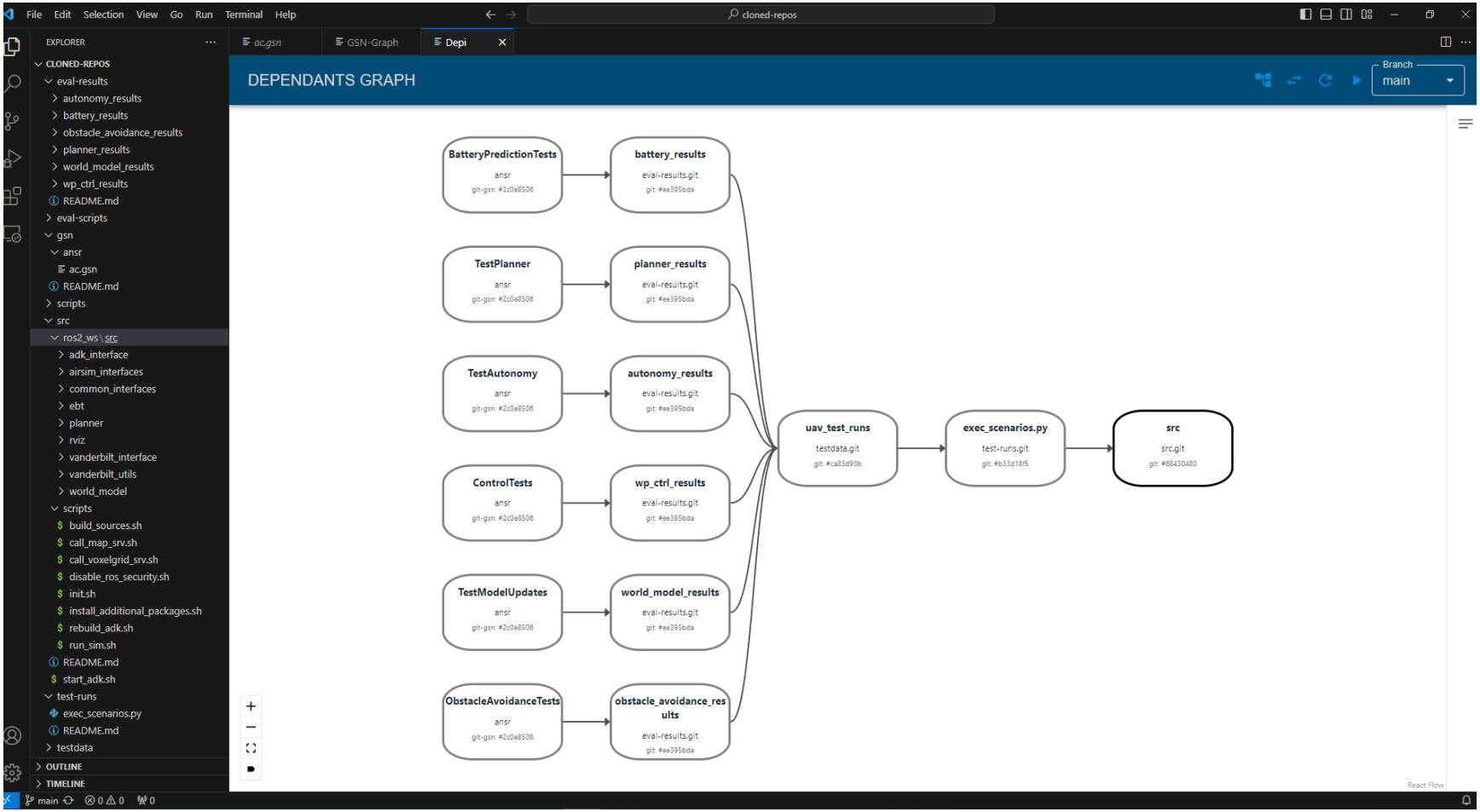
Assurance Provenance Demonstration: Global dependencies



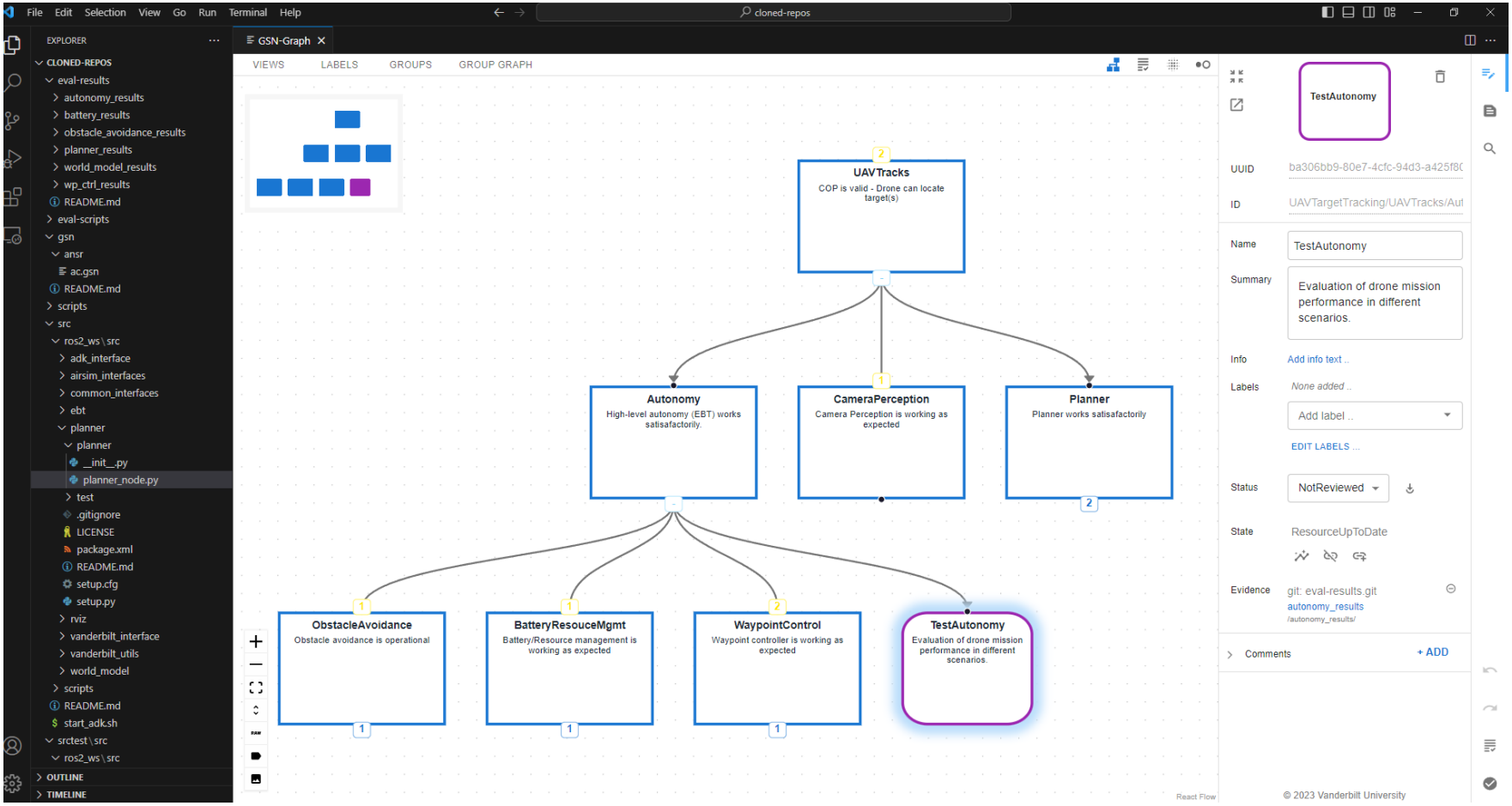
Assurance Provenance Demonstration: Dependency after a change



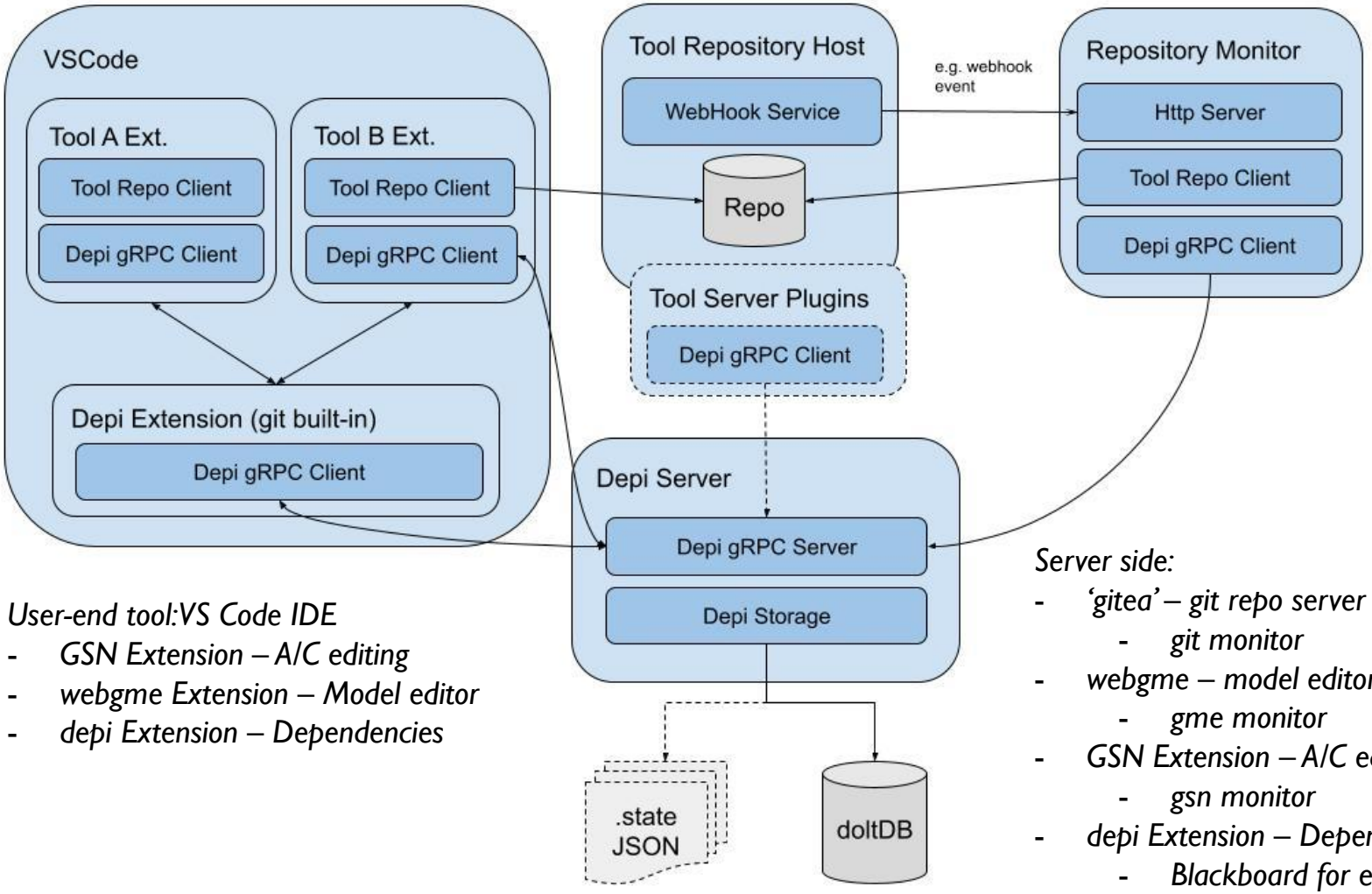
Assurance Provenance Demonstration: Dependencies ‘cleaned’



Assurance Provenance Demonstration: Assurance case evidence



Implementation: Tool architecture



User-end tool:VS Code IDE

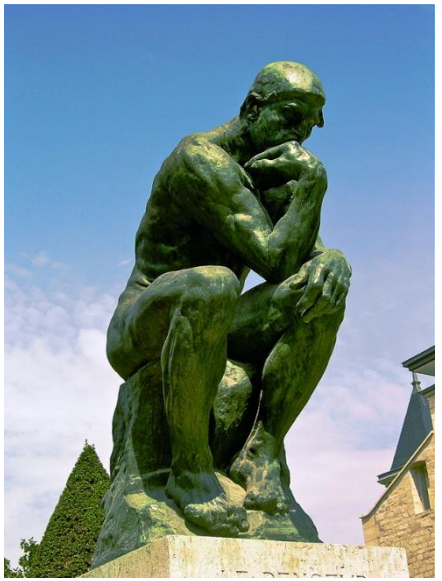
- *GSN Extension – A/C editing*
- *webgme Extension – Model editor*
- *depi Extension – Dependencies*

Server side:

- *'gitea' – git repo server*
  - *git monitor*
- *webgme – model editor server*
  - *gme monitor*
- *GSN Extension – A/C editing*
  - *gsn monitor*
- *depi Extension – Dependencies*
  - *Blackboard for editing*
  - *Database backend*

Summary

- HASS requires complex 'documentation'
  - Models for requirements, specifications, design
  - Implementation: code, tests, tools/settings, docs…
  - Structured assurance arguments + evidence
- Artifacts are linked via complex dependency relations
- Agile development processes necessitate version control
  - Linear/branching versioning + merge,…
- Tooling::
  - Assurance case editor
  - Dependency tracking database
  - Event monitors: git, WebGME, GSN repository, …
  - Server: Linux + docker containers
  - Client: VS Code + extensions
- Challenges:
  - Complexity of relations
  - Management with concurrent updates
  - Continuous Integration/Assurance/Deployment …



A new paradigm for software development where continuous assurance is an integral part of the continuous engineering process?

https://github.com/vu-isis/CAID-tools