DORNERWORKS

AFRL

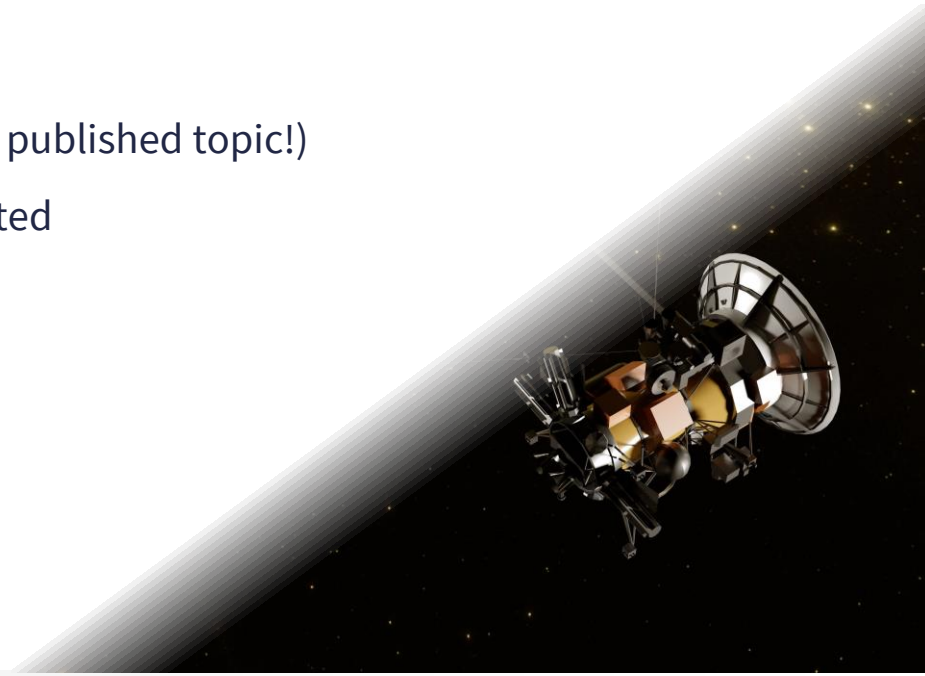
Cyber-Hardened Satellite Software (CHSS) VMM

seL4 Summit 2026
Robert VanVossen

Work funded by Phase II AFRL SBIR
Contract: FA864925P0315
TPOC: Dan Trujillo (joseph.trujillo.4@spaceforce.mil)

Current state of space

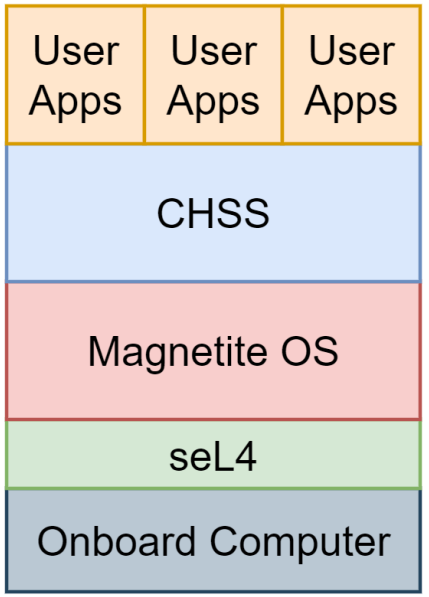
- Many satellites and other space vehicles are using frameworks such as cFS and F’
  - This is great for system re-use!
  - However, these frameworks did not have security in mind when they were developed
- Many security weaknesses are present on these systems
  - Little isolation between applications and services
  - Communications across the systems are fully open (anyone can subscribe to any published topic!)
  - Whenever an application has an issue, the whole OS generally needs to be restarted
- This is a real concern
  - More infrastructure moving to satellites
  - Space is an incredibly contested environment



Cyber-Hardened Satellite Software (CHSS)

An AFRL/RJ Technology

- A paradigm shift of the space industry by designing with security from the ground up
- CHSS provides a cyber-resilient flight software stack on top of seL4 and the Magnetite OS
- seL4 is a formally verified microkernel which guarantees it meets its specification, is free from entire bug classes, and data integrity and confidentiality are maintained.
  - This results in guaranteed software isolation between applications.
- CHSS adds the space vehicle functionality, apps, services for security
  - Secure pub/sub communications with enforced security policy
  - Reusable apps for telemetry, time keeping, tables, command ingest, health and safety, etc
  - Granular restart policies
  - Highly configurable
  - Integrates with user mission SW

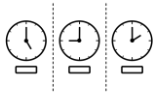


CHSS Security Claims

Recovery: When an application faults, it will restart with little to no impact on other applications.

Dataflow Isolation: There are no unintended software dataflows within CHSS, and all intended software dataflows are subject to the security policy enforced at the Software Data Bus (SDB).

Resource Isolation: Applications are guaranteed access to resources granted to them and are not subject to availability attacks on those resources.



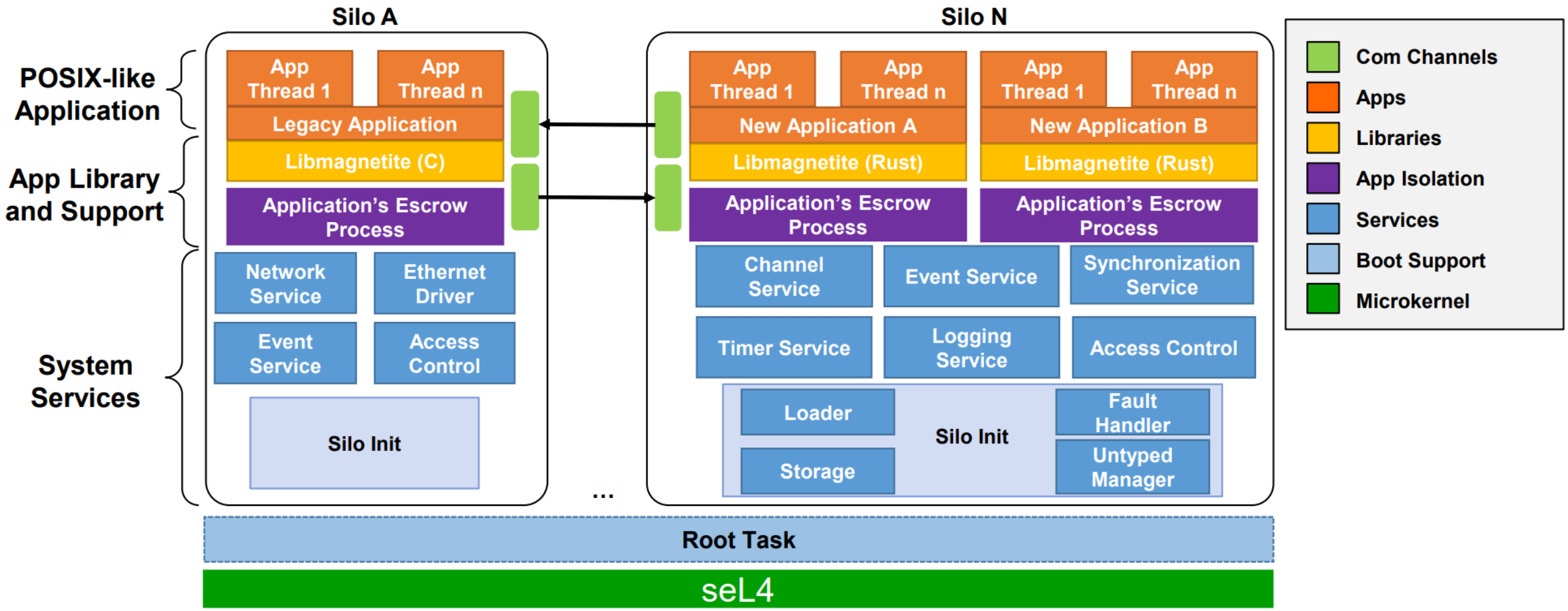
Temporal Isolation: Attacks on application timing requirements can be identified and bounded prior to mission execution.

Implementation Safety: The implementation of CHSS lacks security-relevant classes of incorrectness, such as memory safety, undefined behavior, and arithmetic over/underflows.

Magnetite

- Magnetite OS is written in Rust on top of seL4 and provides services expected by frameworks, such as cFS
  - Rust provides memory and type safety at the OS level
- Previous Presentations:
  - Magnetite: Rust-Based OS Services for seL4 (2023 seL4 Summit, MIT-LL)
  - Porting NASA core Flight System to Magnetite on seL4 (2025 seL4 Summit, MIT-LL)
- Silos (Isolation buckets)
  - CPU allocation
  - Cache-colored Memory
  - Peripheral Allocation (irqs, mmio regions)
  - OS services and applications run in a single silo
  - Communications between silos

Magnetite System



Project Objective

- Our main objective is to make CHSS easier to adopt

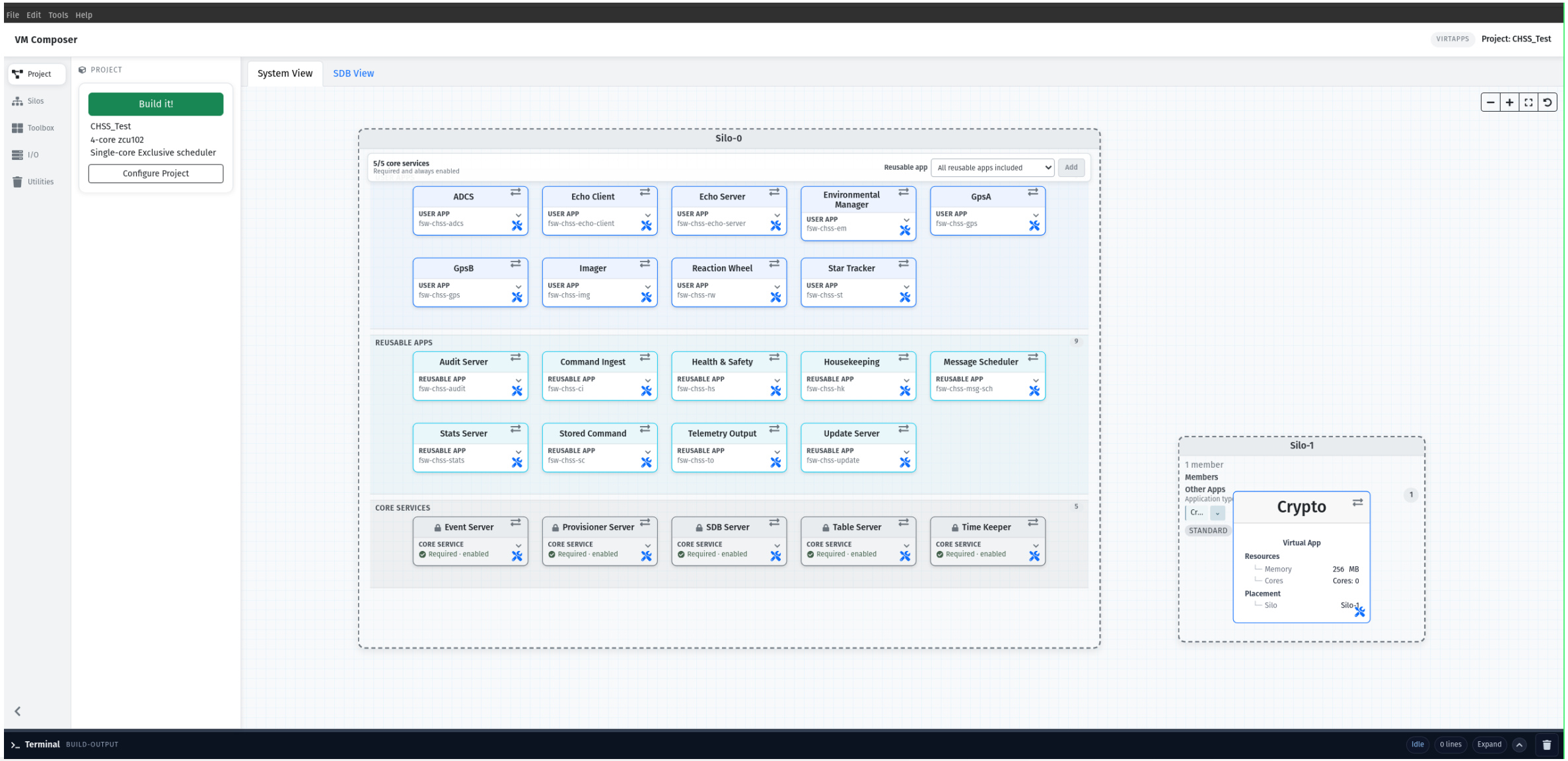
- Two-pronged approach:

1. Add CHSS-awareness to our GUI development tool, VM Composer, allowing for more automated, easier development of missions on top of CHSS.
2. Implement a VMM for CHSS/Magnetite so that legacy mission/flight software can used with CHSS and be iteratively integrated with/ported to the CHSS environment.

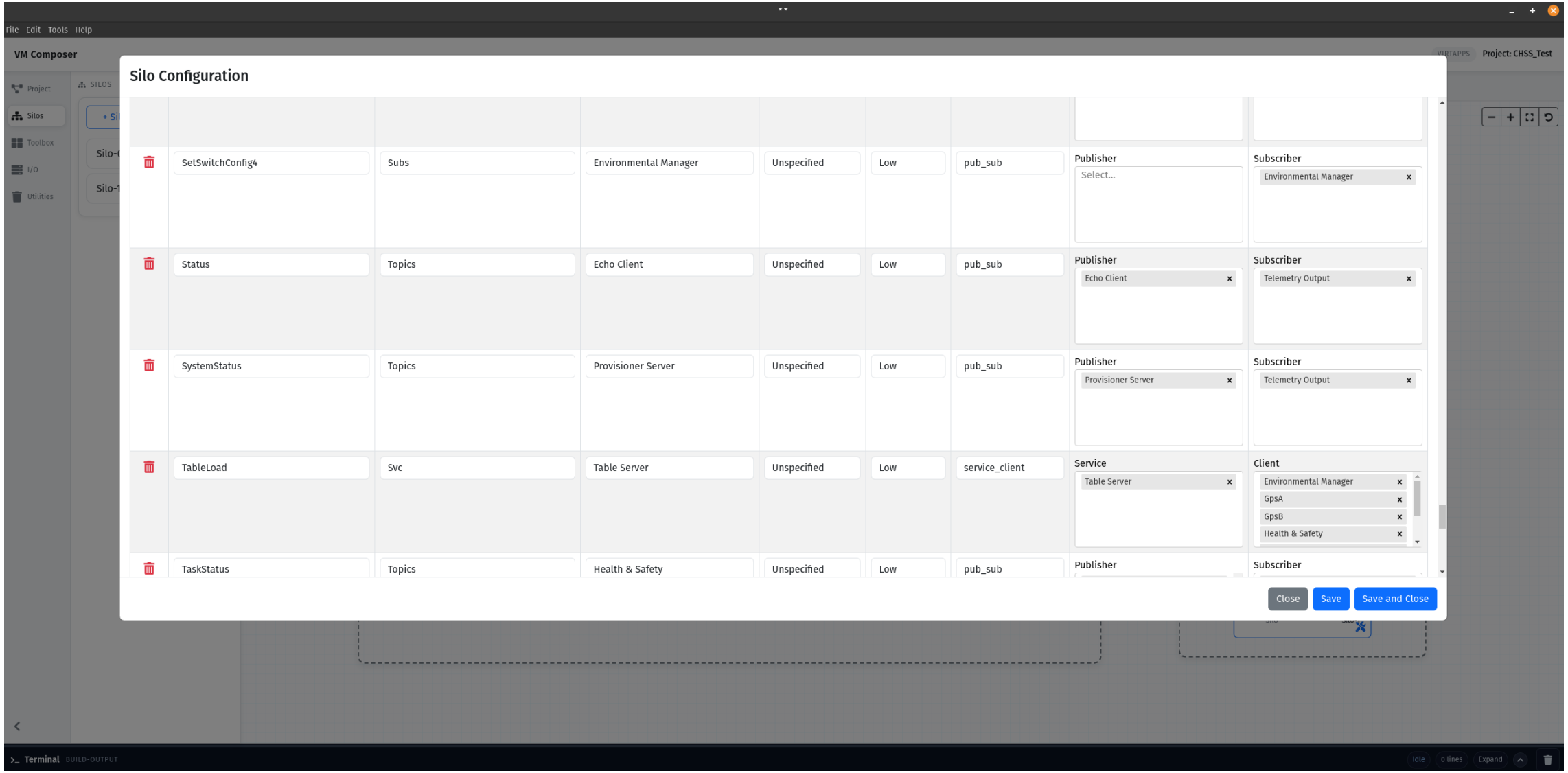
VM Composer

- Previously a mission system integrator needed to write a mission crate from a template or from scratch
  - Required quite a lot of code and deep understanding of the CHSS subsystems
- For this effort, VM Composer was made CHSS-aware
  - Allows for configuration of systems on CHSS (Configures Magnetite, CHSS apps, User apps, and VMMs)
  - VM Composer now generates a majority of the mission crate
- Current Gaps:
  - Some manual coding of mission crate still required
  - More automation of initial configuration/knowledge of expected topics
  - More automated analysis options to improve certification processes

VM Composer: CHSS Configuration

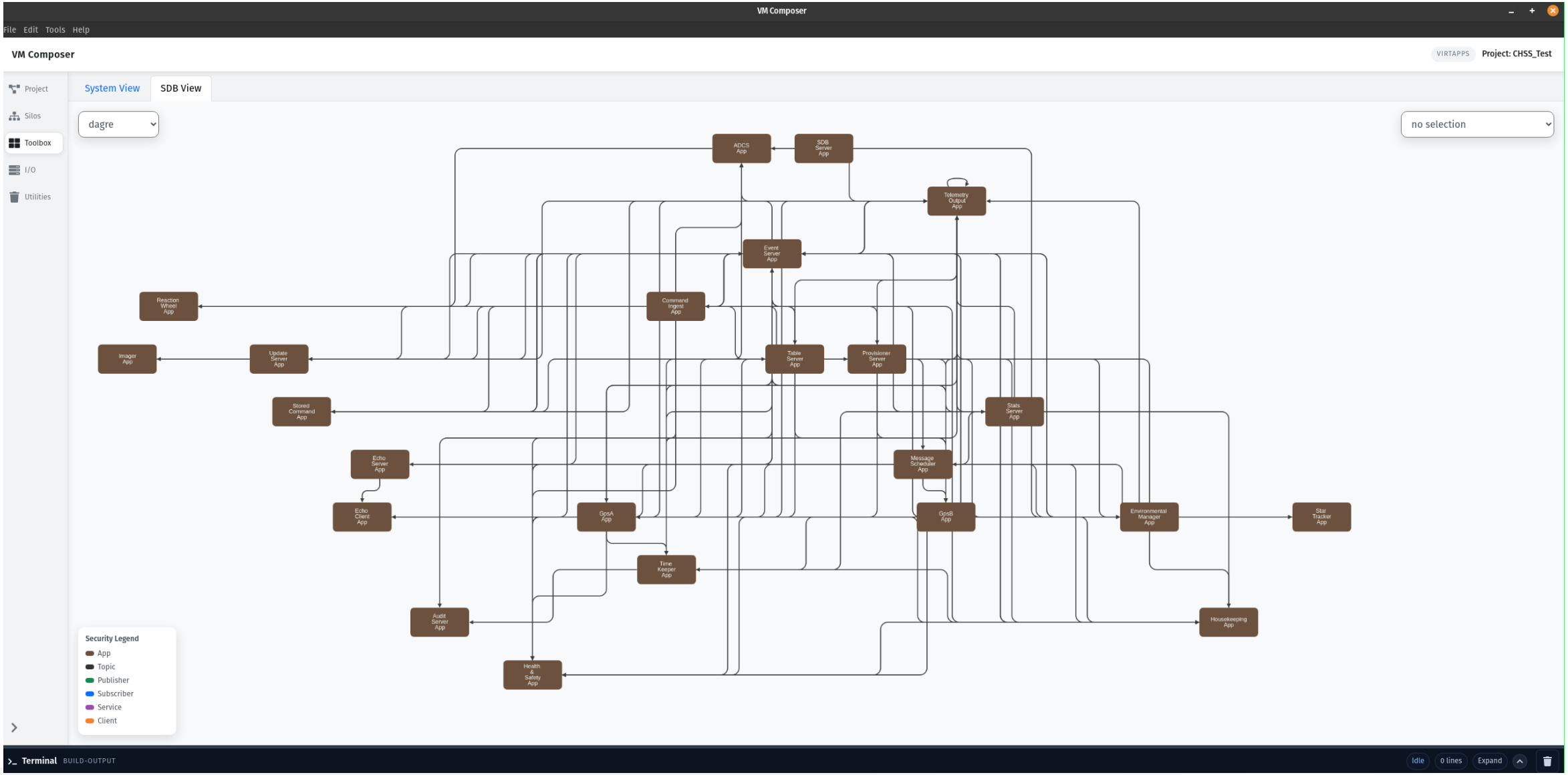


VM Composer: CHSS Comms view



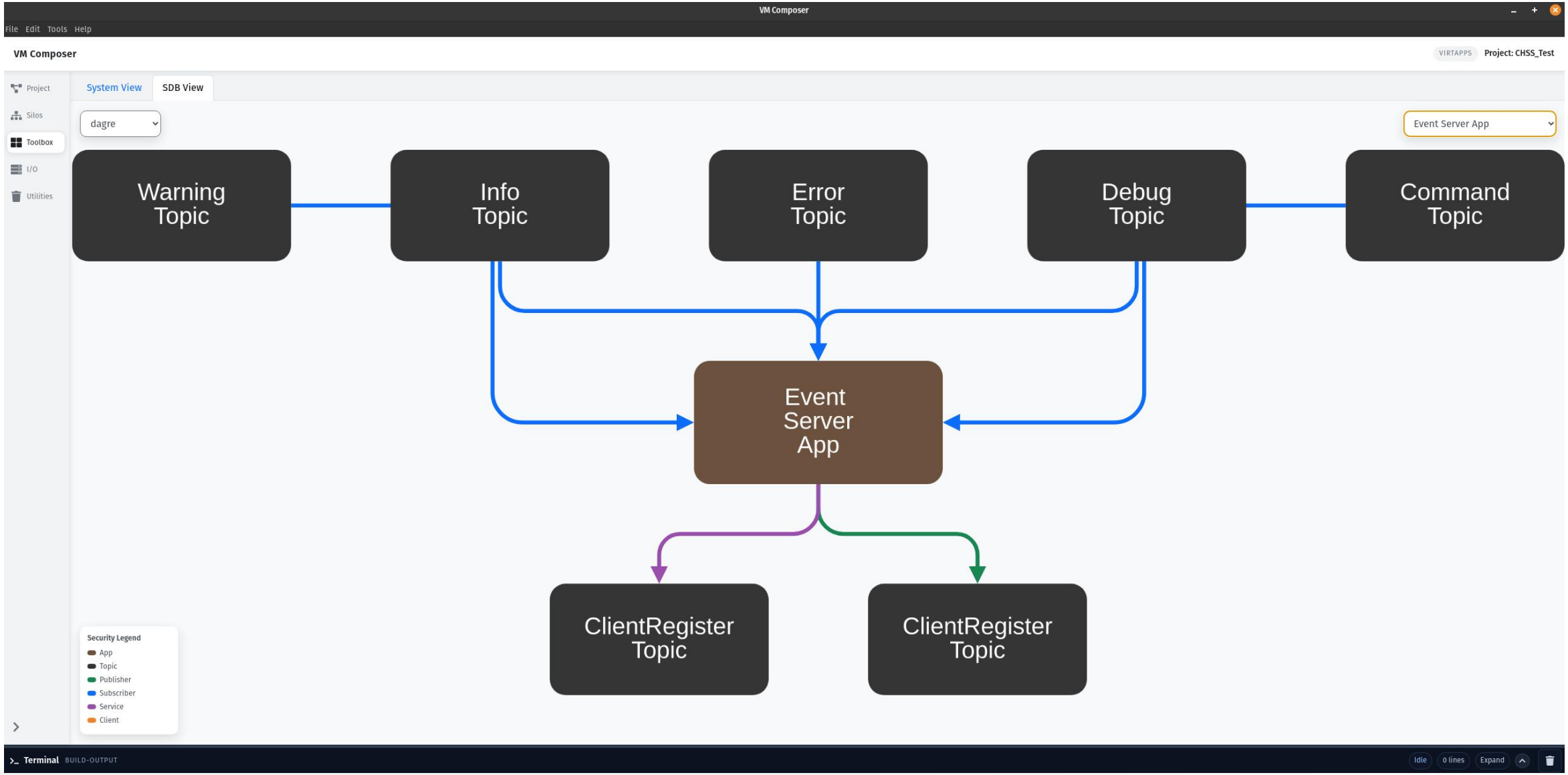
DORNERWORKS AFRL

VM Composer: CHSS Comms view

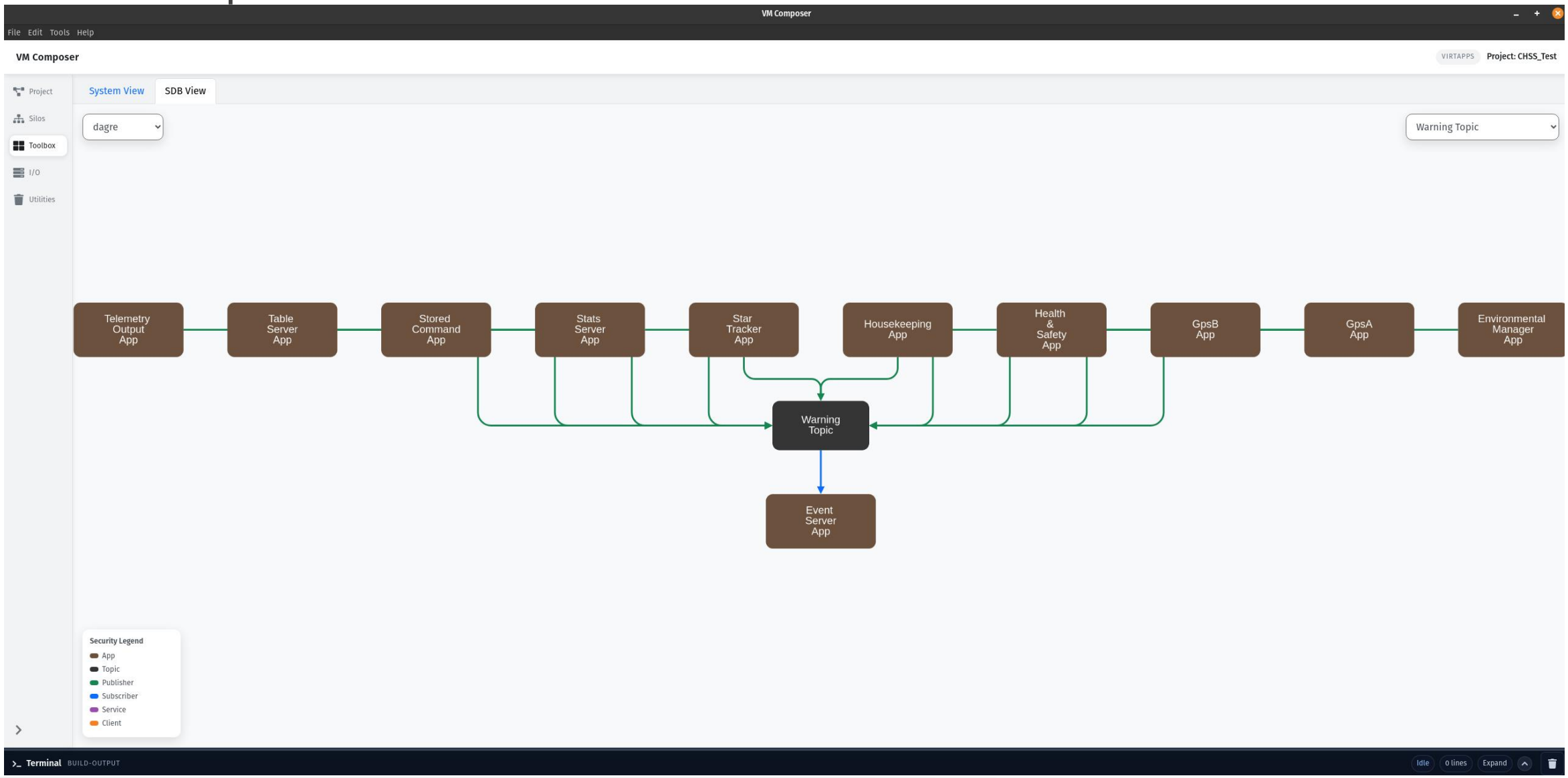


DORNERWORKS AFRL

VM Composer: CHSS Comms view

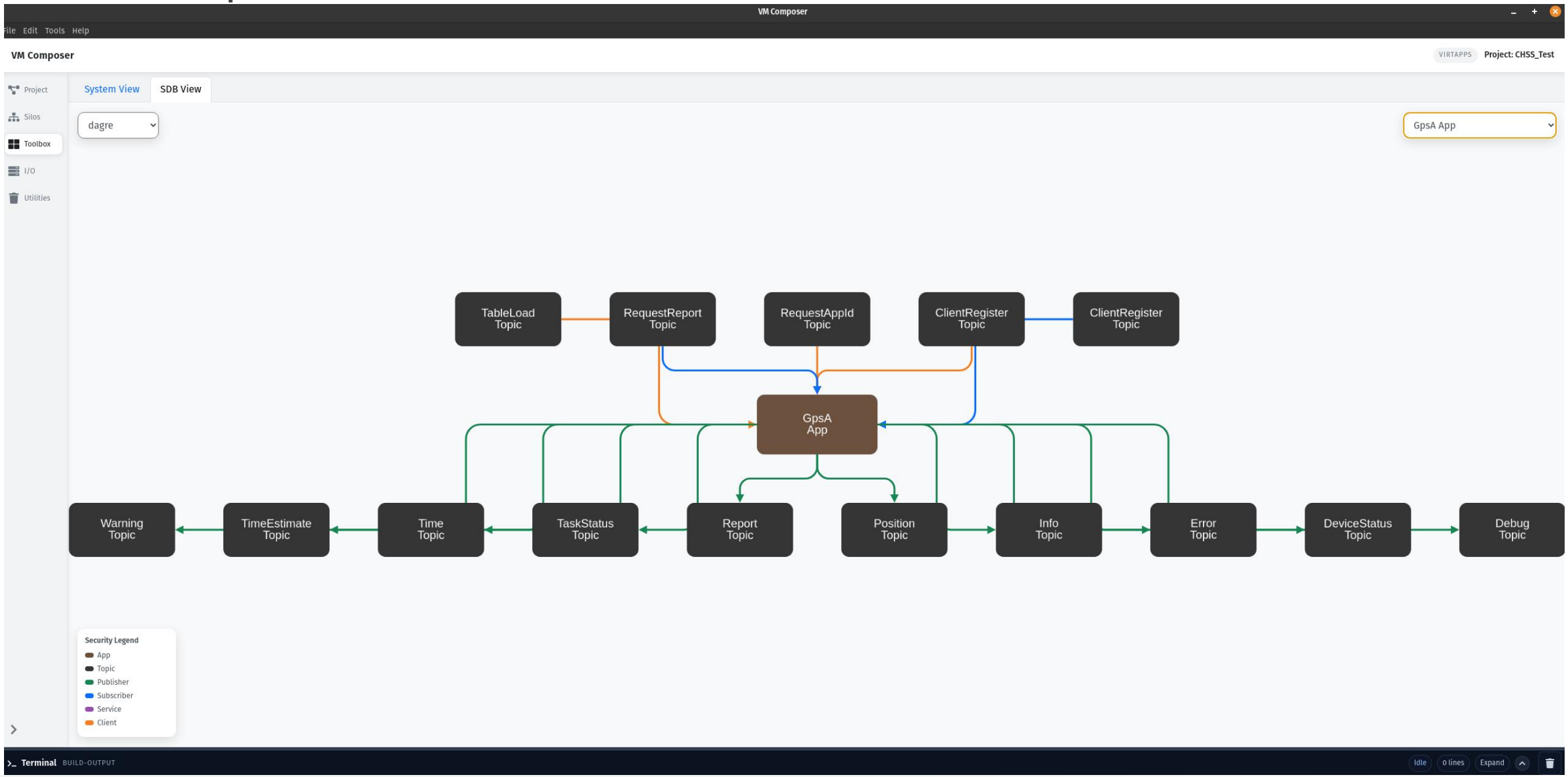


VM Composer: CHSS Comms view



DORNERWORKS AFRL

VM Composer: CHSS Comms view



VMM

- Written as a Magnetite service that can run alongside CHSS
- Written in Rust for ARMv8 platforms
  - Continues providing memory safety into the VMM
  - Roughly ported some components from the camkes_vm and libvmm
- Some interesting design considerations for rust
  - Emulated devices (UART, Interrupt controller) need an owner (The VMM server)
  - Various components need to inject or Ack IRQs, but the emulated interrupt controller is no longer global
    - Fault handlers for emulated devices return Option<IrqAck> and Option<IrqInject> objects, when appropriate
      - This tells the VMM server whether to forward on these calls into the emulated interrupt controller
  - Defined a trait for emulated devices and store them in a Vec<Box<>>. This allows an arbitrary number of emulated devices that must provide a known API
    - This makes it easy to know how to implement new devices and what to provide.

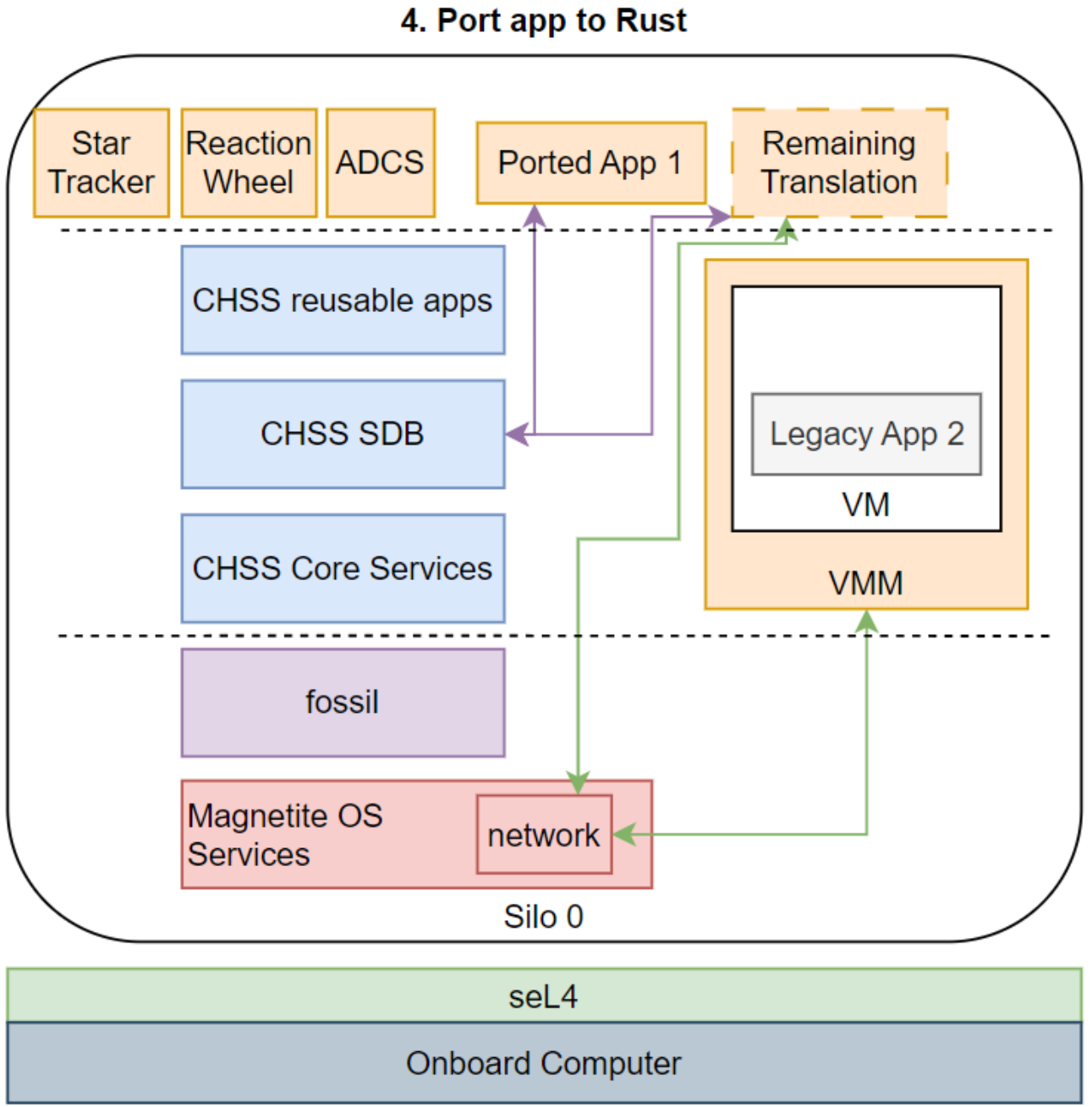
VMM

- Utilizes what is already provided by Magnetite to produce a much smaller VMM
  - Capability management, memory management, notifications, filesystem access, logging, etc
- Some pieces are quite generic, such as seL4 VM fault types, emulated devices (VGICv2 & v3, PL011 Uart, PSCI), SMC management, and VCPU management.
  - Working to open these up as a general, rust-based VMM library that could target other frameworks or OSes built for seL4, such as Microkit
- Have currently run Linux guests on multiple ARMv8 platforms
- Next steps
  - add virtio interfaces so the VM can connect to things like the Magnetite filesystem and network
  - DMA-device pass-through

Iterative Development Process

- A Cyber retrofit development process

1. Take legacy system and run it in a VM alongside CHSS system
2. Implement a CHSS app that translates data between CHSS and the VM
3. Split translation CHSS app into one for the app currently porting and the remaining comms
4. Port legacy application to Rust in CHSS
5. Repeat 3-4 until all critical legacy apps have been hardened



Iterative Development Process

- This process replaces all-or-nothing approaches which require all legacy code to be ported before the system can be fielded.
- At each completed step in the process, the system is runnable and deployable, allowing for interim releases
- The entire legacy codebase does not need to be ported to CHSS
  - Less critical SW can remain in the VMM for deployment
- One remaining challenge: lots of legacy software was not written in a modular way, making it difficult to identify and split out critical parts of the system into their own applications.

Conclusions

- CHSS is the next step in space vehicle development, providing security and cyber resiliency from the ground up
- CHSS-Aware VM Composer makes it much easier to develop, modify, and audit a CHSS-based system
- The Magnetite VMM allows developers to quickly integrate their legacy flight/mission software with CHSS
  - It also allows for an iterative porting process that facilitates hardening the most critical legacy applications in a way that results in always having a deployable system
- Working to provide a public version of the VMM to make transition to the CHSS VMM more seamless

Questions

- Email: Robert.VanVossen@DornerWorks.com