Scalable booting for System-on-Chip in ATLAS L1CT

Giulio Muscarello

ATLAS Level 0 Central Trigger for Phase 2

Many more SoCs foreseen

Currently: 13 ATCA boards (lab + experiment)

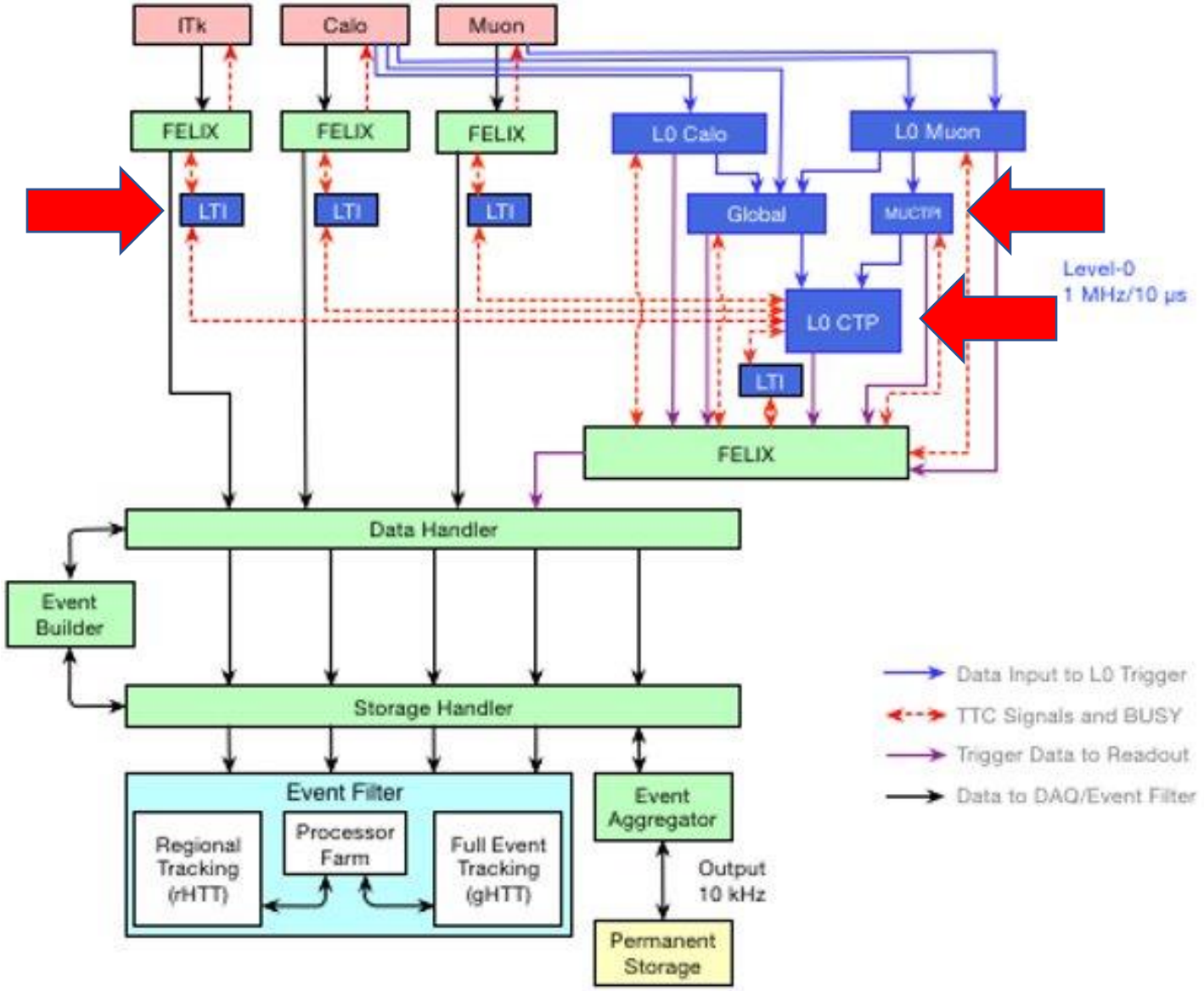
8 MUCTPI, 5 LTI, 2 custom Krias

Phase 2: est. 50 ATCA boards in the experiment

1 CTP, 2 MUCTPI, 48 LTI

SoC@CERN: 2k SoCs!

Approx. 50% CMS, 25% ATLAS



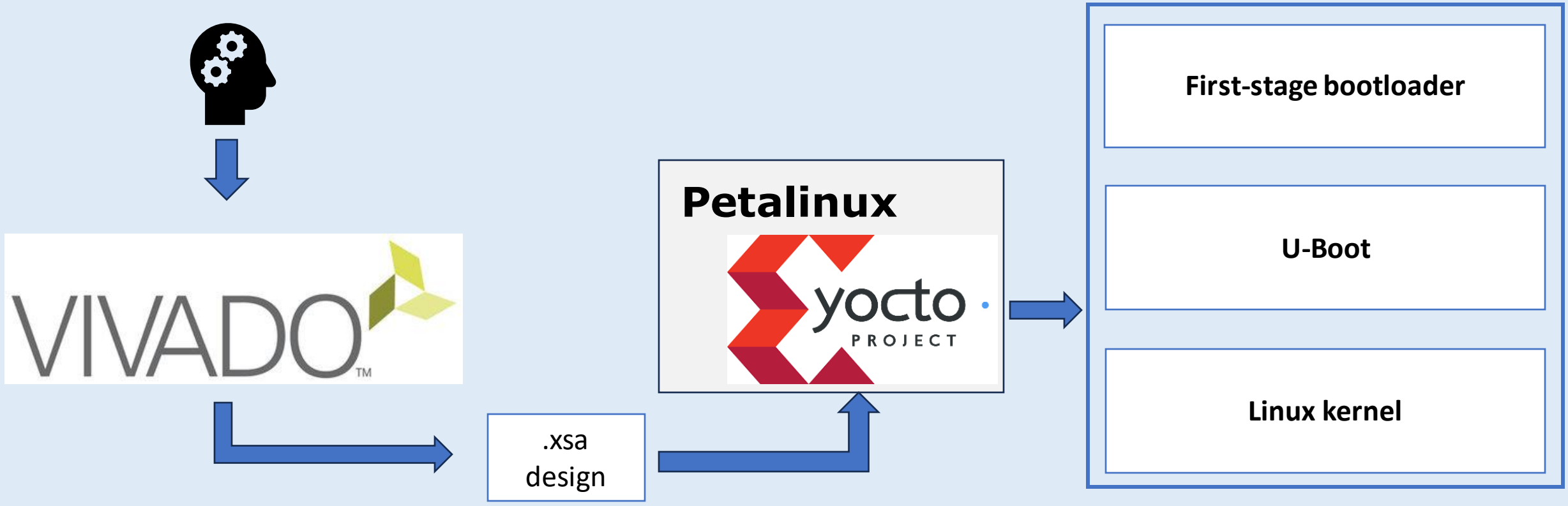
SoC workflow

This seminar will touch three aspects:

- **Bootloader and kernel compilation**
  Building with a common base (same CPU) and different hardware "flavors"
- **Booting**
  Determining what kernel to download, where to read the filesystem from, etc.
- **Host configuration**
  Keeping list of users, software packages, etc. up to date

Bootloader compilation

Ideal toolchain



The problem of external hardware

Problem: the boards have both on-chip and "external" devices

- **On-chip devices** (CPUs, clocks, etc.)
  Vivado knows about them, can generate the correct instructions for the bootloaders/kernel/etc.
- **On-board devices** (connected via buses like I2C, SPI, SGMII, etc.)
  Vivado doesn't know that e.g. you connected an EEPROM on I2C!
  You need to somehow supply this information to the bootloader and Linux

How does the system know what to initialize?

- On desktops, ACPI/PCI automatically enumerates devices
- On SoCs, a **device tree** must be built in or loaded

Device tree

A device tree is a **description of a hierarchy** of devices.

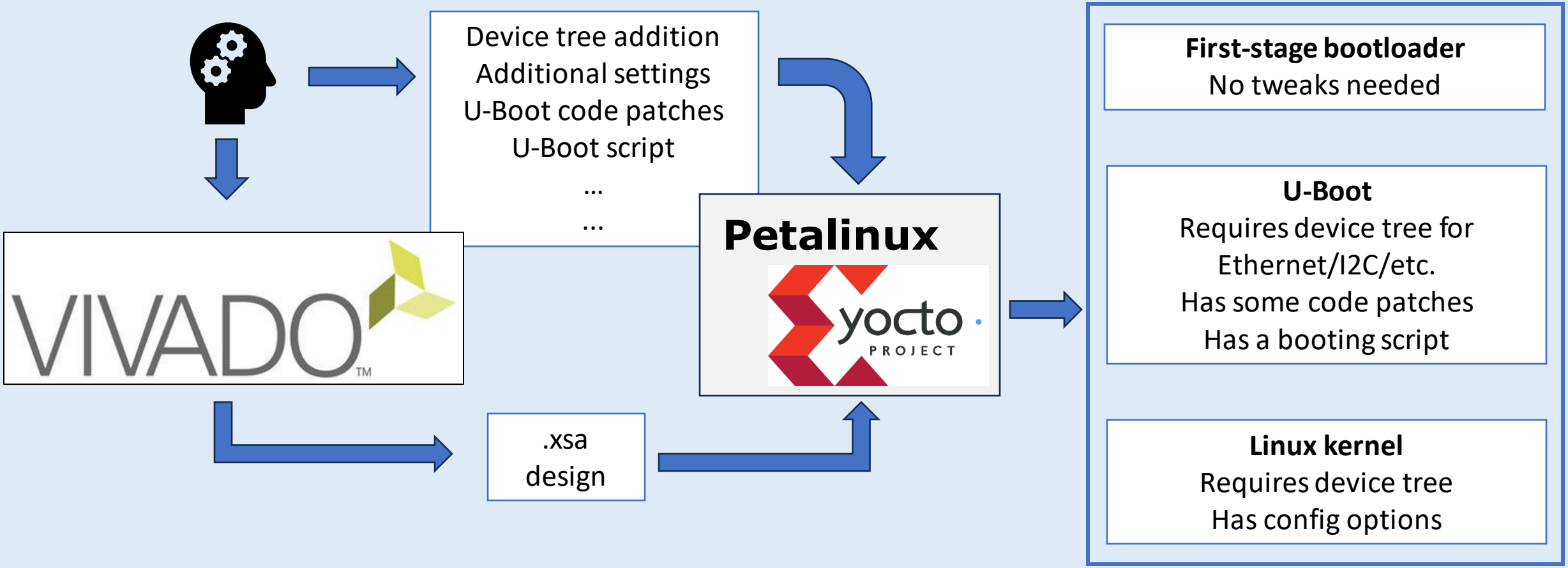
Each device typically has:

- A class (I2C controller)
- An address, whether in memory or on a bus (0xff020000)
- A **compatible driver** (Cadence driver)
- Other **properties**, e.g. interrupts
- Can contain **children**, especially if it is a bus

Vivado generates a base device tree, to which we add external buses and devices.

```
// I2C controller
i2c0: i2c@ff020000 {
    compatible = "cdns,i2c-r1p14";
    status = "okay";
    interrupt-parent = <&gic>;
    interrupts = <0 17 4>;
    // An I2C multiplexer at address 0x74
    i2c-mux@74 {
        compatible = "nxp,pca9548";
        i2c@0 {
            // On channel 0, at address 0x54, there's an EEPROM
            mac_eeprom: eeprom@54 {
                compatible = "atmel,24c08";
            };
        };
    };
};
```

Actual toolchain



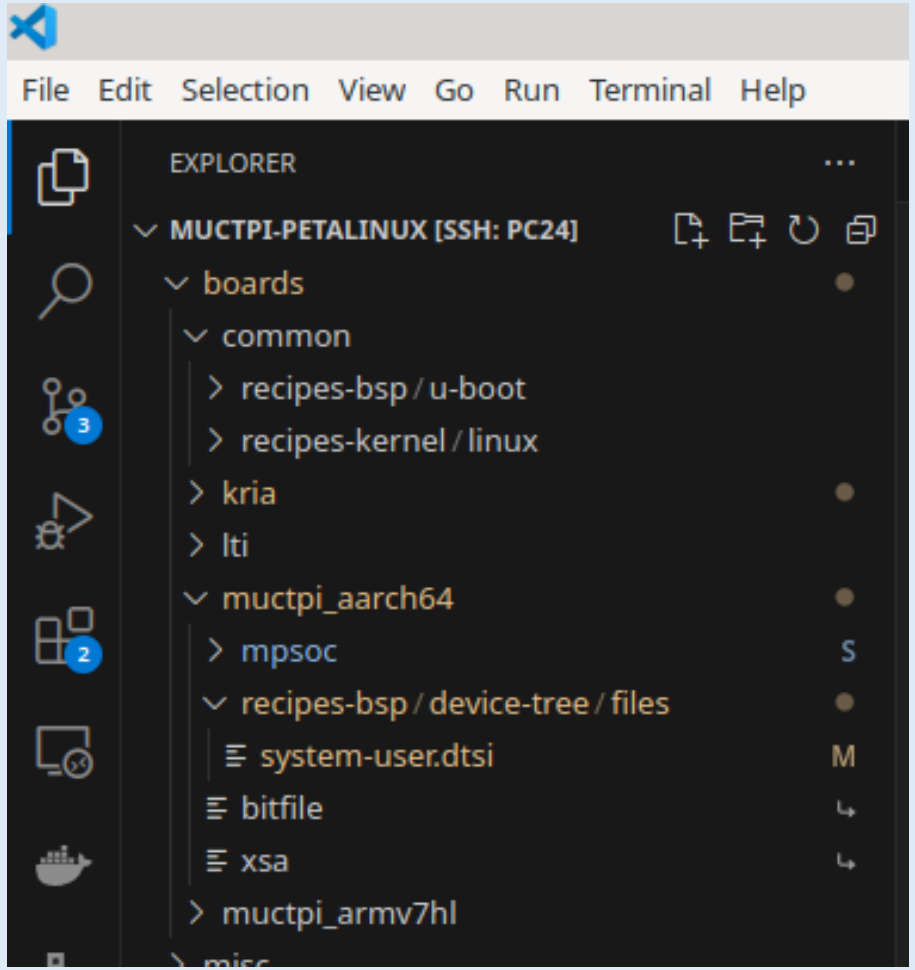
Scaling to several types of boards

The workflow for each board type is mostly the same, but there are small differences due to external hardware, internal quirks to be patched, etc.

How do we manage these **similar-but-not-identical workflows**?

- One repository per board type: *lots* of duplication, things go out of sync :(
- Common repository with **board-specific patches**: minimal duplication, centralized workflow :)

See our actual workflow on the right: "`boards/common`" contains the shared U-Boot and Linux build, and "`boards/muctpi`" has the board-specific device tree



soc/petalinux-template

A Petalinux workflow hosted on GitLab that:

- Starts from a basic, CPU-only template
- Applies some "shared" patches (eg. adding features to the bootloader)
- Applies board-specific patches
- Builds the project automatically

Adding a new type of board is "just" a matter of writing the board-specific patches: the rest comes for free!

soc/petalinux-template

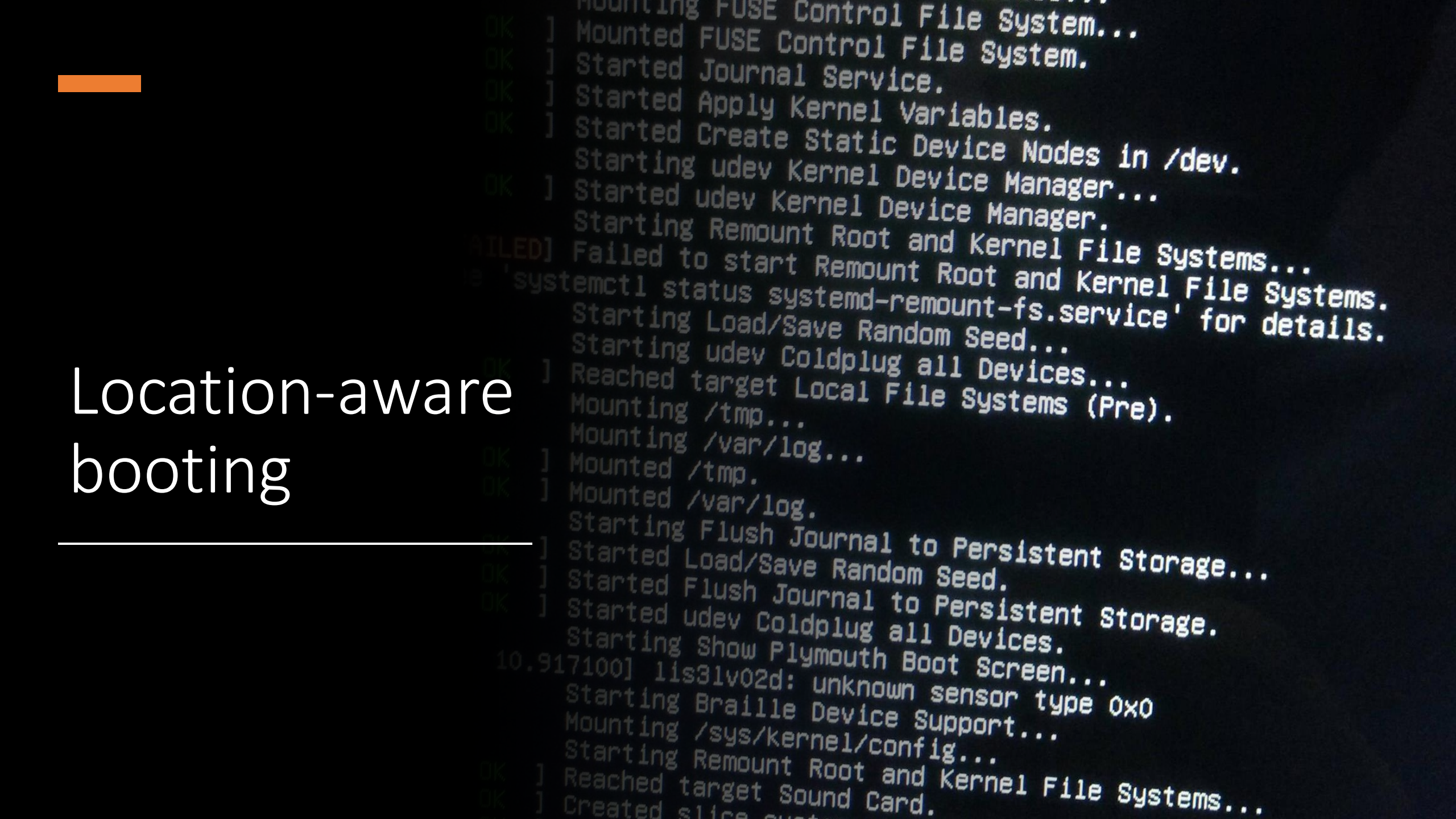
The template enhances **cooperation**: the SoC I.G. provides a baseline image with common features for the **benefit of all**, and individual groups fork it and add their own patches (or alternatively cherry-pick features)

Currently in the template:

- IPMC communication
- DHCP Client ID in U-Boot

Recently open sourced – you're welcome to **try it out** and give feedback!

Location-aware booting



Booting overview

After initializing the hardware, we want to download the kernel and use a remote filesystem so we can update more easily, not have storage requirements on the SoC, etc.

1. **Hardware initialization**
   This is done "locally", you just copy the boot files to internal flash/SD card
2. **Network setup**
   Esp. in the experiment: how does the board identify itself?
3. **Kernel download and startup**
   How does it select the correct kernel? How is it updated?
4. **Remote filesystem mount over NFS**
   Where does the filesystem live?

Where is this logic implemented?

- Currently in U-Boot, a "simple" bootloader by Xilinx
- Sysadmins want to move to GRUB, a general-purpose bootloader

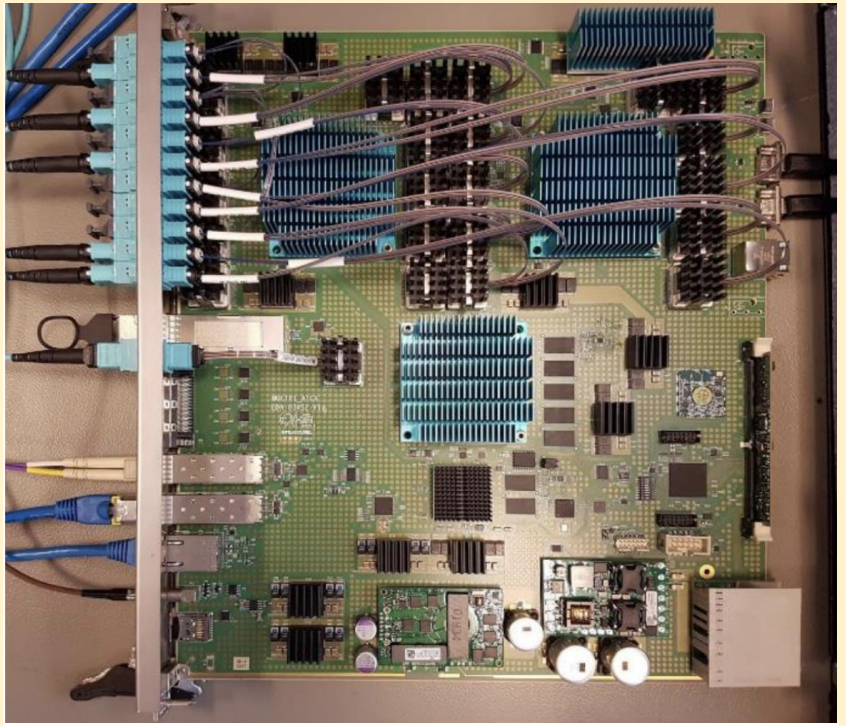
Identifying boards

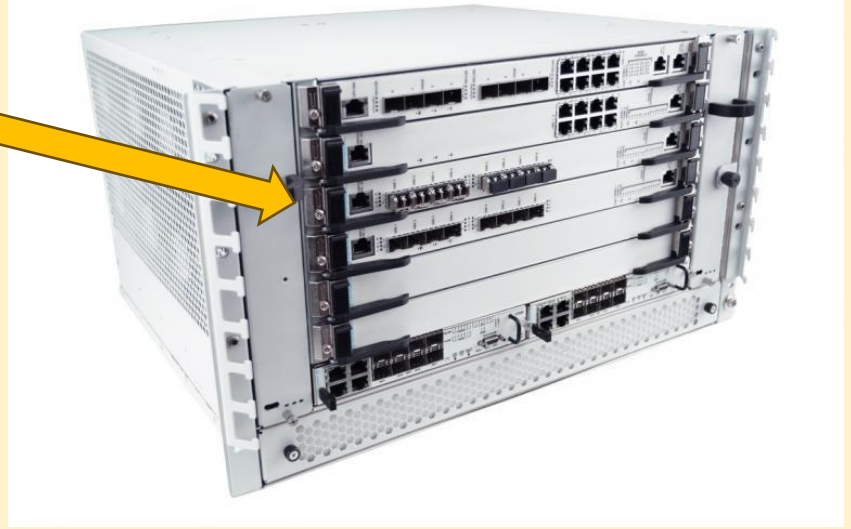
How do you serve the correct configuration to a board?

- In labs/interactive usage, the user cares about a specific, unique board: **board identity**
  MAC address X is of type "LTI" and needs settings "ABC"
- In the experiment, we rely on shelf location: **shelf identity**
  Shelf X, slot Y is of type "LTI" and needs settings "ABC"

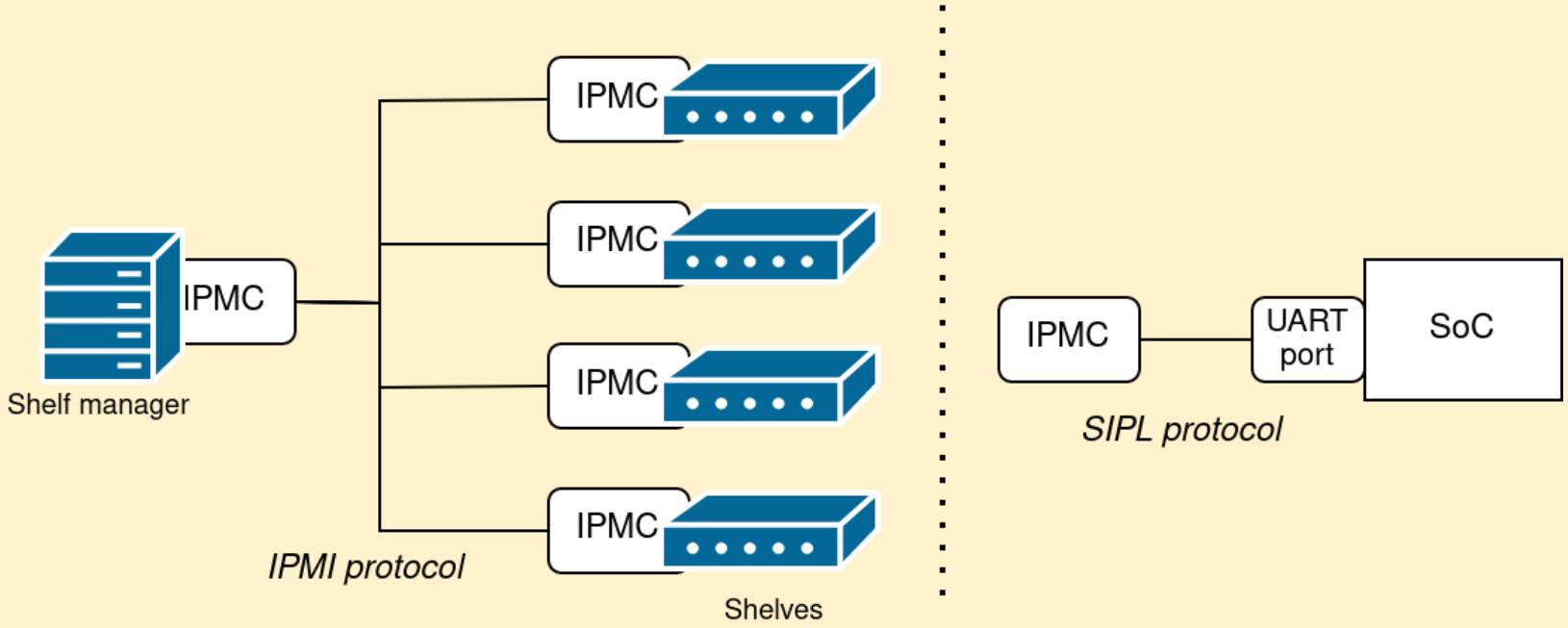
Board identity via MAC is well understood and used, no problem here

But how is shelf identity read? And how is it used in identification?





ATCA shelf



A **shelf manager** is a powerful tool that we won't cover here: it controls power, sensors, fans...

Key point: the IPMC exposes a serial protocol (SIPL) for getting information and executing commands

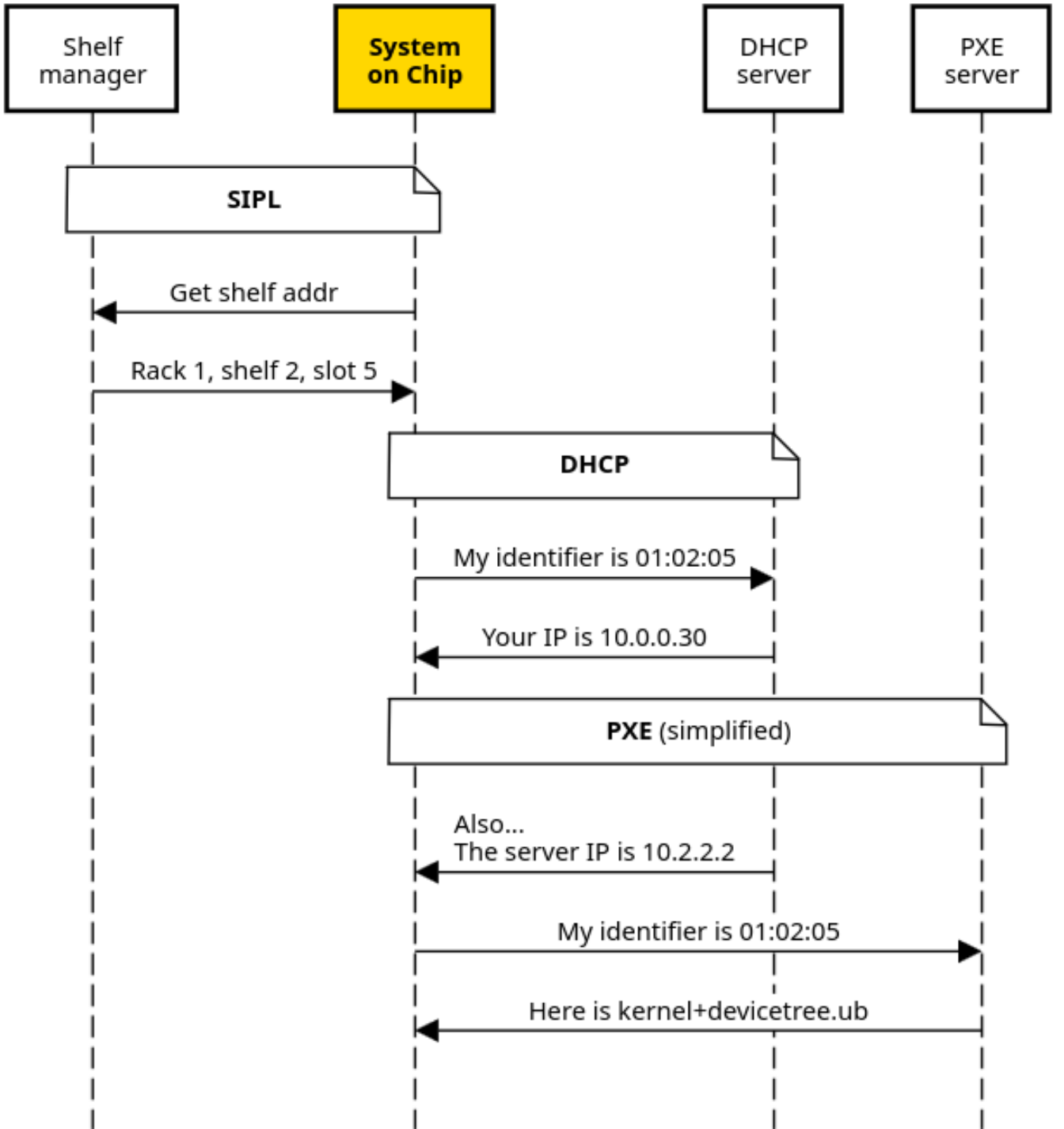
The bootloader can figure out the board's location with a serial message

Booting sequence

1. **Bootloader**: hardware initialization
2. **DHCP**: identifies with central servers, receives network configuration
3. **PXE**: receives booting configuration
4. **Linux**: mounts file system, runs software

Looks a lot like **ATLAS netbooted** nodes:
can reuse existing knowledge and infrastructure, makes sysadmins happy

DHCP Client ID is used to encode shelf identity – varying levels of support

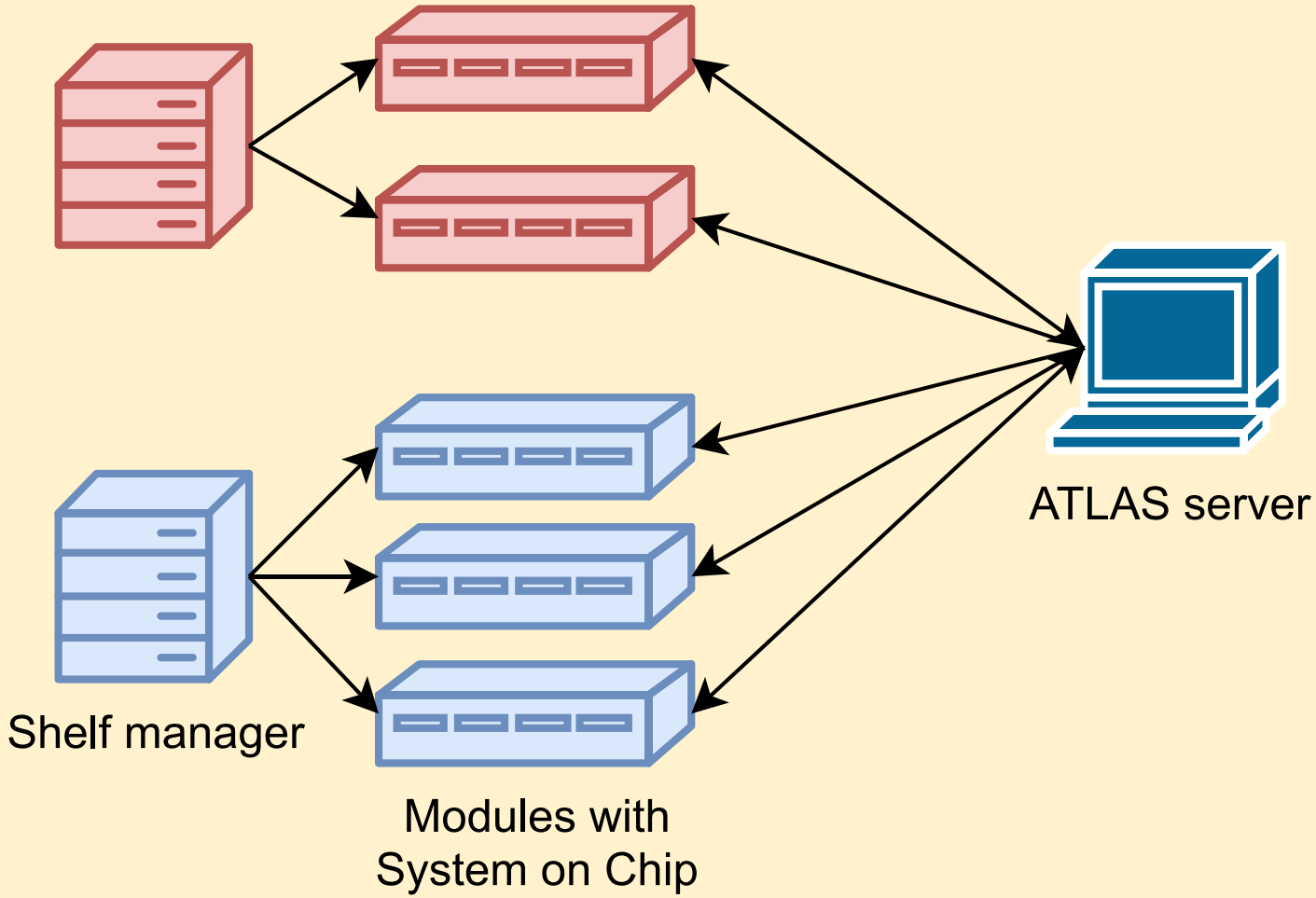


The case for location-aware booting

In short, **shelf identity determines configuration**

Very flexible:

- **Scales up** to 1000s of devices thanks to easy classification of nodes
- **Scales down** to single boards when needed
- Can **hot-swap** boards



Location-aware booting in ATLAS L1CT

As of today:

- **Shelf identity** with **DHCP** is enabled on all boards
  DHCP Client ID is supported in the lab-run DHCP server but not with CERN/ATLAS DHCP
  Works reliably for us, but requires to run our own DHCP
  Marc Dobson from SoC Interest Group is in talks with CERN IT for Client ID support
- **PXE** is enabled on Phase-2 devices
  Works well with a local DHCP server, looking into running on GPN
- Phase-1 devices use a legacy mechanism (uEnv.txt)
  Reason: replicate what is in the experiment. Evaluate PXE during next YETS

Overall: satisfied with DHCP+PXE as it is a more standard configuration

Thanks to DHCP+PXE we can **automate** booting configuration (in our case, from a central CSV description of devices)

Host configuration

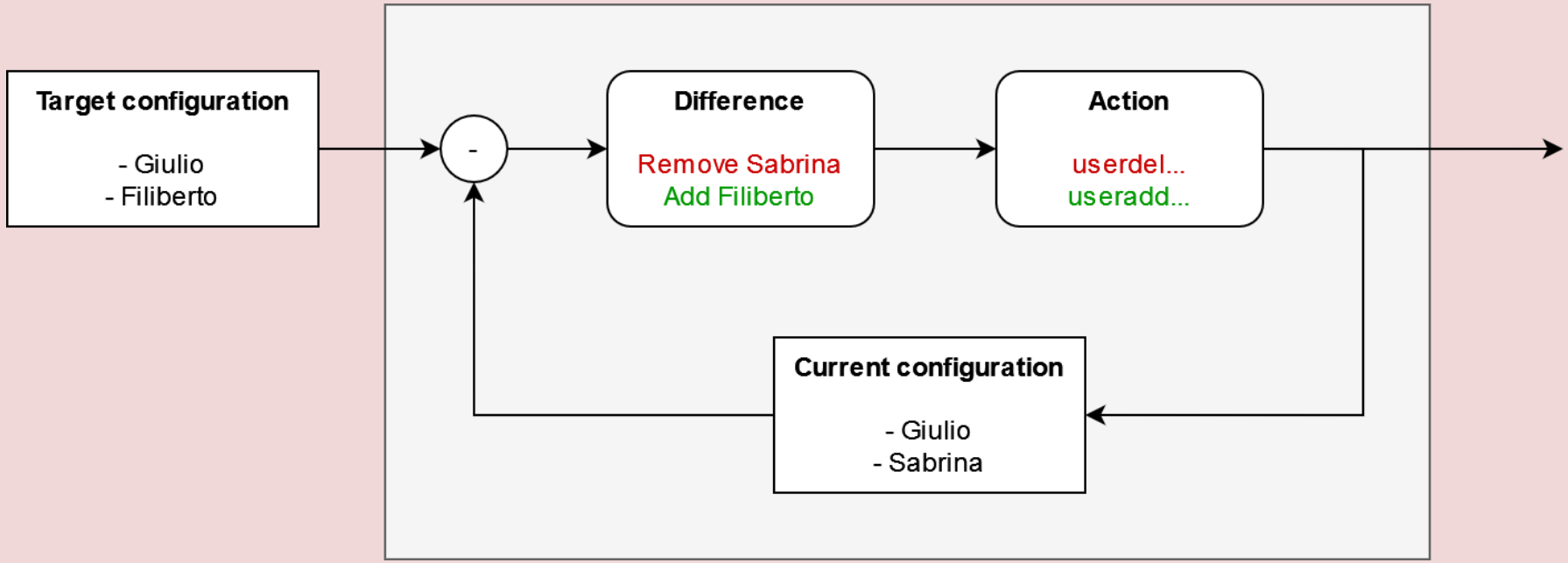
Host configuration

We create a filesystem image on Git containing **users**, TDAQ services and **settings**, etc.

Deploying updates is non-trivial:

- **Overwriting** a live system is risky and convoluted
  Eg. systemd services need to be reloaded/restarted, udev has its own reload procedure
- **Rebooting** with a new image works and is very simple
  But users don't like rebooting (for good reason!)
- Either way, you overwrite local modifications
  Problematic in the lab, not so much in the experiment

The case for declarative configuration



Would be nice to just **declare your target** configuration and let some tool handle the rest
Something like a closed loop control: compare target with current state and act on difference
This paradigm is called **declarative** (as opposed to imperative, open-loop control: e.g. shell scripts)

Host configuration

Specify **what you want**, not **what to do**

Puppet detects differences and applies the necessary changes

- **Simpler specification**
  How would you rewrite the config on the left as a *robust* script?
- **Simpler updates**
  Safe* to run on a live system, no matter how dirty/out of date
- **Greater visibility**
  `--dry-run` will compare states and tell you why the host is misconfigured

```
user {
  'gmuscare':
    ensure     => present,
    home       => '/home/gmuscare',
    shell      => '/bin/bash',
    uid        => 151154,
    gid        => 1307,
}
ssh_authorized_key {
  'my-ssh-key':
    ensure     => present,
    user       => 'gmuscare',
    type       => 'ssh-rsa',
    key        => 'AAAAB3NzaC1...'
}
```

Host configuration in L1CT

Thanks to Puppet we manage **centrally** and **dynamically**:

Users, authentication	Udev rules	FPGA bitfiles	TDAQ configuration
IPMI sensor collection	Memory monitor	Uptime monitor	Jupyter

… on all boards, in the lab. In the experiment we want a "good" base image

Puppet has worked very well for bringing systems to a **common, working state**
We can preview changes thanks to "dry runs": ideal for the lab
Not always obvious how to integrate with systemd, **more experience required**

Outlook

- **Test** our changes: what can break? How can it be made robust?
- Gather **experience** on how to do things "the right way" and **exchange information** with teams using similar tools
- Cooperate with ATLAS and CMS sysadmins on a **common scheme** for Phase 2
  - UEFI (see presentation by Quentin)
  - Split FIT image into a common kernel and a device tree
- Improve **integration** with CERN infrastructure
  - ATCN: Phase 2 devices will not require a bastion host
  - LDAP authentication: use "CERN credentials"
  - Icinga2 monitoring
  - What else? Gather **feedback** from users
- **Open source** our work to other SoC users

Conclusions

- Scalable systems must be **highly automated**: replace/resize/etc. seamlessly
- Industry-**standard** solutions address our needs
- Experimentation in lab successful: we can **iterate faster** and more efficiently
- We are still gathering **experience**

Questions?