Bringing order to chaos with $IDL_NAME

Standardize integration, not formats and protocols

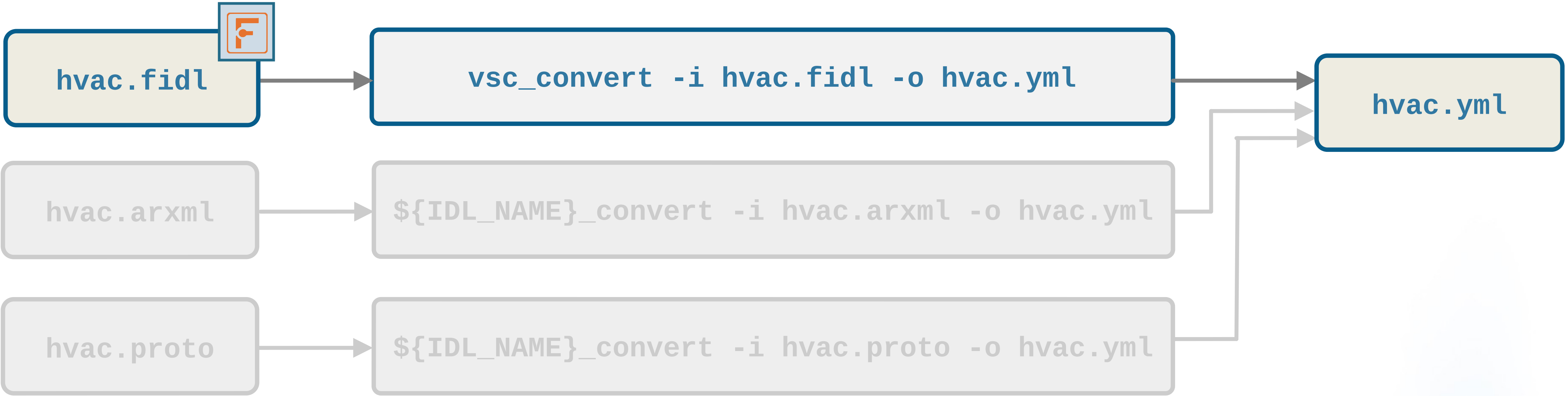
GENIVI Workshop   July 12021

How can we describe a set of vehicle- and connected services that use Adaptive, OpenAPI, DBUS, etc, as a single, uniform system?

Why a new IDL?

- Is it really a new IDL? Or is it just Franca + intermediate/internal YAML representation. (discussion today)
- Capture all automotive systems (vehicle to cloud) needs in one IDL
- Reuse Franca IDL principles – a lot of thought went into it, especially to cover all types of interfaces
- Creation of a new, **flexible and lightweight tool suite**
- The establishment of "the standard data model" is heavily leaning towards being started from **VSS**. Therefore:
  - Create a solution for services that **fits well with VSS**
  - Mimic formats and principles (YAML based) of VSS to achieve a **consistent development environment**
- Import standardized services from GENIVI and other sources. This requires, at minimum, agreeing on the IDL for that representation.
- Promote gradual industry movement towards a common language for describing program/subsystem behaviors
- Capabilities you will get:
  - Auto-export your services to $IDL_NAME to leverage open source tool chains and code generators
  - Export service descriptions for your existing features to $IDL_NAME to leverage...
  - Auto-generate protocol bridges between vendor-provided services and existing in-vehicle network stacks
  - Bind ARXML to gRPC, Protobuf IDL to SOME/IP, FrancaIDL to OpenAPI, etc.

Where is Franca IDL in all this?


hvac.fidl
vsc_convert -i hvac.fidl -o hvac.yml
hvac.yml
hvac.arxml
${IDL_NAME}_convert -i hvac.arxml -o hvac.yml
hvac.proto
${IDL_NAME}_convert -i hvac.proto -o hvac.yml

- Franca IDL is the main supported input format
- Bidirectional translation between Franca IDL and Vehicle Service Catalog formats is fully supported
  - VSC could be seen as a YAML variant of Franca IDL with no loss of information
  - VSC potentially drives some improvements to Franca IDL → next step in Franca evolution?
- Additional formats supported as needed

“What about OpenAPI”

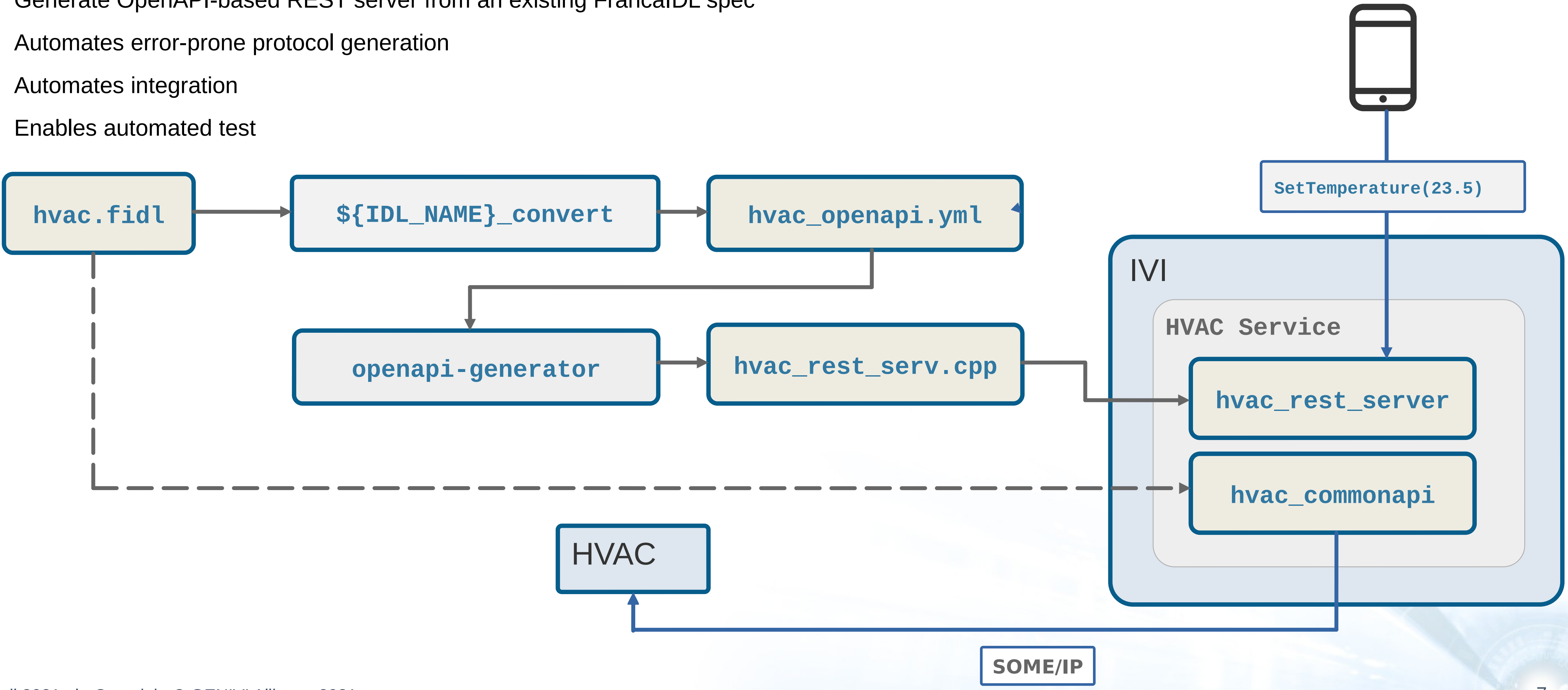
- OpenAPI is optimized for REST*ful* HTTP interfaces
- $NEW_IDL (and Franca IDL) is flexible for **all types** of interfaces
- We are aiming for a **generic, single, main** IDL
- BUT! OpenAPI should be part of the development ecosystem (convert to/from existing interface descriptions, leverage OpenAPI tools)

“What about AsyncAPI”

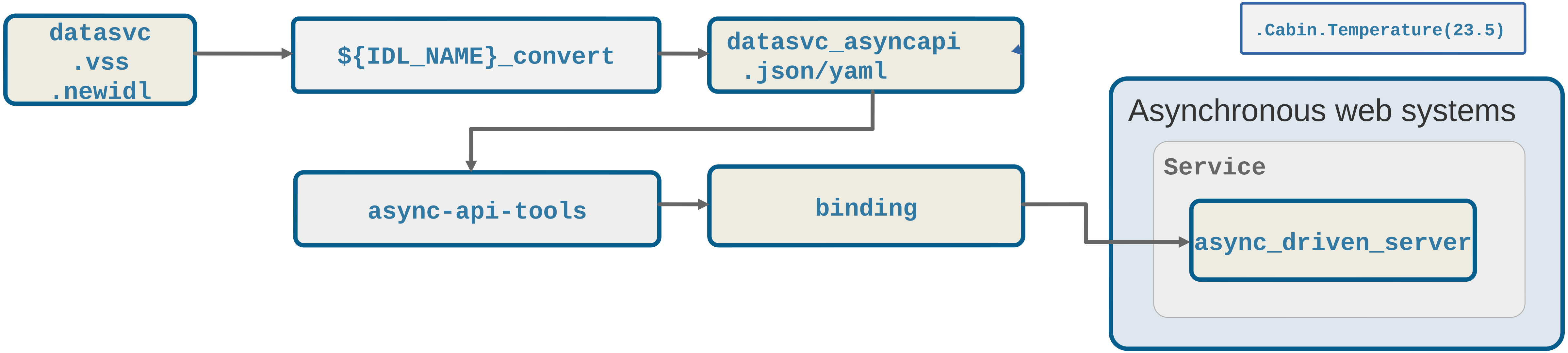
- AsyncAPI describes publish/subscribe data exchange interfaces, not generic method calls, etc.
- $NEW_IDL (and Franca IDL) is flexible for **all types** of interfaces
- We are aiming for a generic, single, main IDL
- BUT! AsyncAPI should be part of the development ecosystem
  (convert to/from existing interface descriptions, leverage OpenAPI tools)
- (Being pub/sub focused, it is rather a discussion for the VSS signal ecosystem)

Integration example FrancaIDL - OpenAPI

- Generate OpenAPI-based REST server from an existing FrancaIDL spec
- Automates error-prone protocol generation
- Automates integration
- Enables automated test

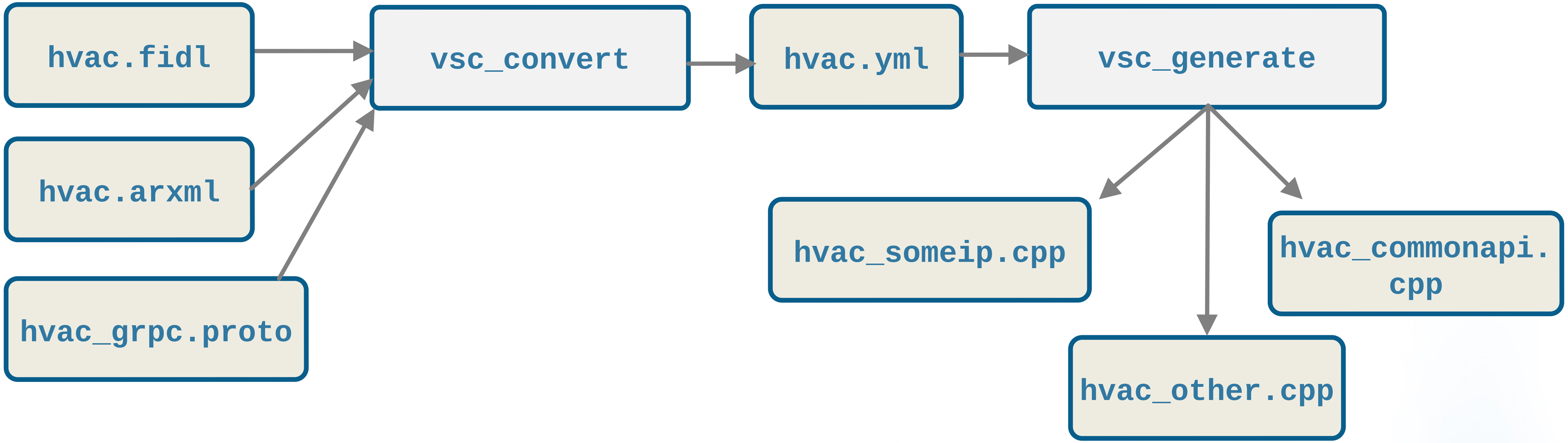


Integrating AsyncAPI



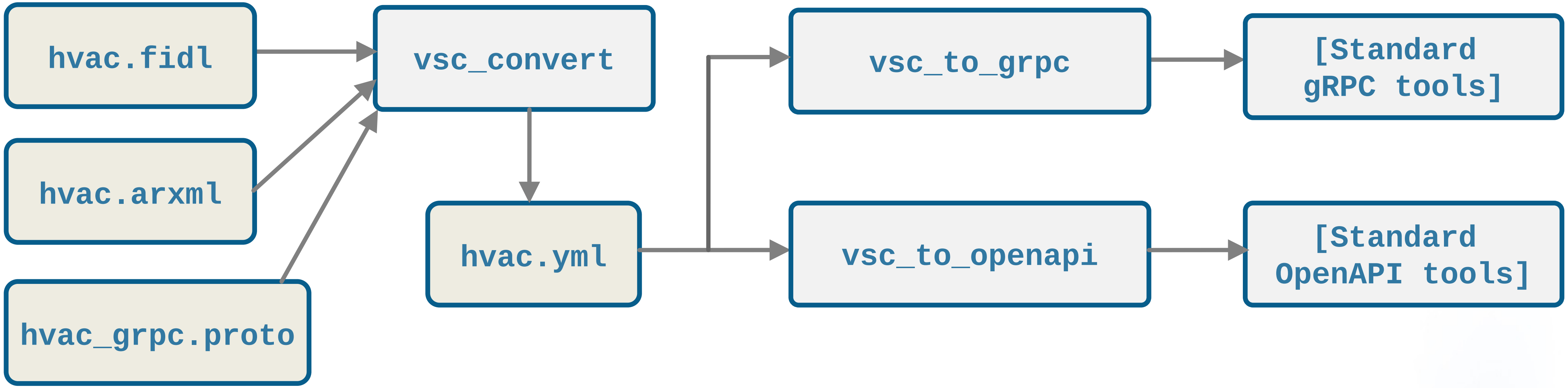
- Convert descriptions of services to AsyncAPI to leverage AsyncAPI code generation / binding tools
- Use AsyncAPI for documentation generation?

Generating in-vehicle code



- Conversion tool normalizes existing interface specs to VSC YAML format
- Code generation tool creates automotive-targeted stub code
- Links to CommonAPI and other existing stacks

Generating non-vehicle code



- Single IDL specification used as single source of truth that is fed into automated tool chains
- Normalized VSC specification can be converted to multiple other target IDL formats
- Standard target tooling used to create stub code for non-vehicle deployment

Common Vehicle Interface Initiative (CVII)
Alignment recap

- **CVII** drives the automotive industry conversation around alignment of core standards and technologies
- **CVII Tech Stack** assumes *data-model* and *services-model* commonality have been or will be achieved in other tracks
- **CVII Tech Stack** selects/aligns on a *reasonable number* of protocols and technology bindings

Approach:

1) ? “Develop” a full-featured IDL with heavy influence from existing choices, Franca IDL in particular

2) Provide tools/bindings to core technologies with a *simple and extensible* approach

3) Create conversions to/from other choices *where appropriate*:

   - To ensure smooth migration
   - To ensure efficient leverage of existing ecosystems and implementations

4) Promote movement over time towards *industry-standard* IDL

5) Avoid *everything-to-everything* conversion approach, which simply continues fragmentation

Strategic and methodical avoidance of the **XKCD standards effect**

6)

(*Google it if by any chance you need to)

Corporate adoption strategies

- Keep existing IDL specification library as to minimize disruption
- Normalize IDL specification library to VSC format to leverage conversion and code generation tools
- Normalize vendor-provided IDL spec to VSC format to leverage same tool chain used by internal specs
- Use IDL spec to write automated tests that can validate multiple service protocols (gRPC, SOME/IP, etc)
- Optionally, develop new specifications directly in VSC format, gradually retiring original IDL format

Open questions

- Are all of you on board with the creation of a common IDL strategy?
- Which format to we specify the service catalog in - VSC or FrancaIDL?
- Which input formats to we need to support in addition to FrancaIDL?
- Which output format do we need to convert to?
- Which target protocols do we need to generate code for?
- ...

1)

END OF DECK
The rest are backup slides

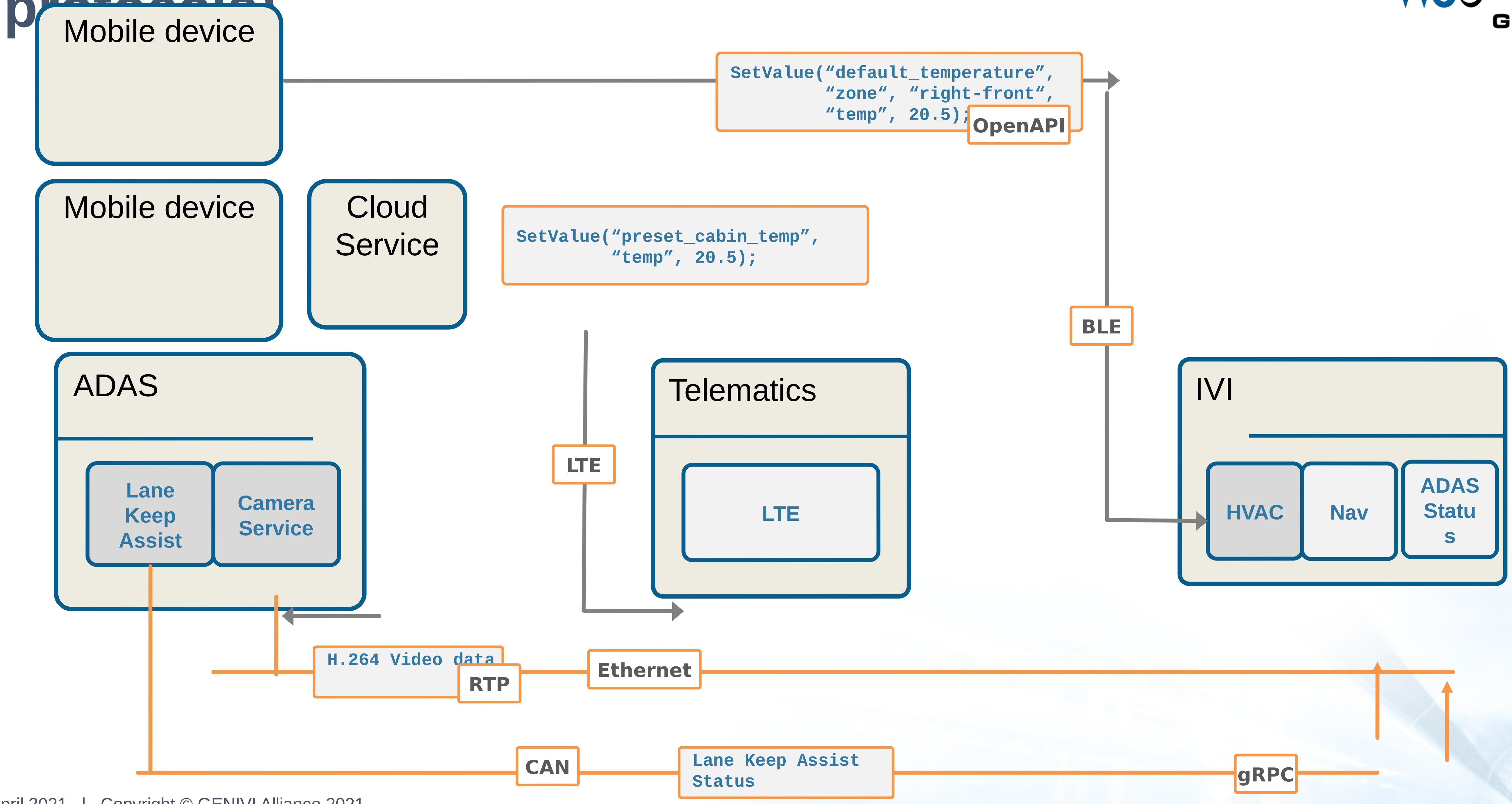
Market drivers to standardize services

- **OEM drivers**
  - Use standardized APIs to decouple solutions from vendor-specific technologies
  - Push for standard-compliance in RFIs & RFQs to ease side-by-side bid comparison
  - Use open source, standardized tools, and joint industry effort to create a higher starting point, allowing programs to focus resources on brand-differentiating experiences

- **Tier 1 & 2 drivers**
  - Implement standardized API to minimize program customization and maintenance, migrating toward off-the-shelf offers to OEMs
  - Portal/Host value-added services from third parties

- **Non-automotive drivers**
  - Manage mixed-asset fleets with minimum of system integration and maintenance
  - Widen and accelerate market for new 3rd party automotive services

Typical Deployment (showcase variety of protocols)



```
SetValue("default_temperature",
         "zone", "right-front",
         "temp", 20.5);
```

```
SetValue("preset_cabin_temp",
         "temp", 20.5);
```

```
H.264 Video data
```

```
Lane Keep Assist
Status
```

Vehicle Service Catalog Objectives

1. **Specify a GENIVI catalog of standardized services to drive industry transformation to Service Oriented Architecture**

2. **Specify a W3C protocol for vehicle access**

3. **Promote industry adoption of standardized services and protocols**

Why Yet Another Standard?

- **Language and protocol agnostic**
  We need to try out different languages, protocols, and philosophies before we commit to something we want to standardize

- **Scale across 100s of interoperating services**
  Name spacing, interface imports, deployment models, and API vs. Implementation version management are all needed in large-scale deployment

- **Lightweight**
  CLI oriented. Five minutes to running tutorial. Small, componentized codebase

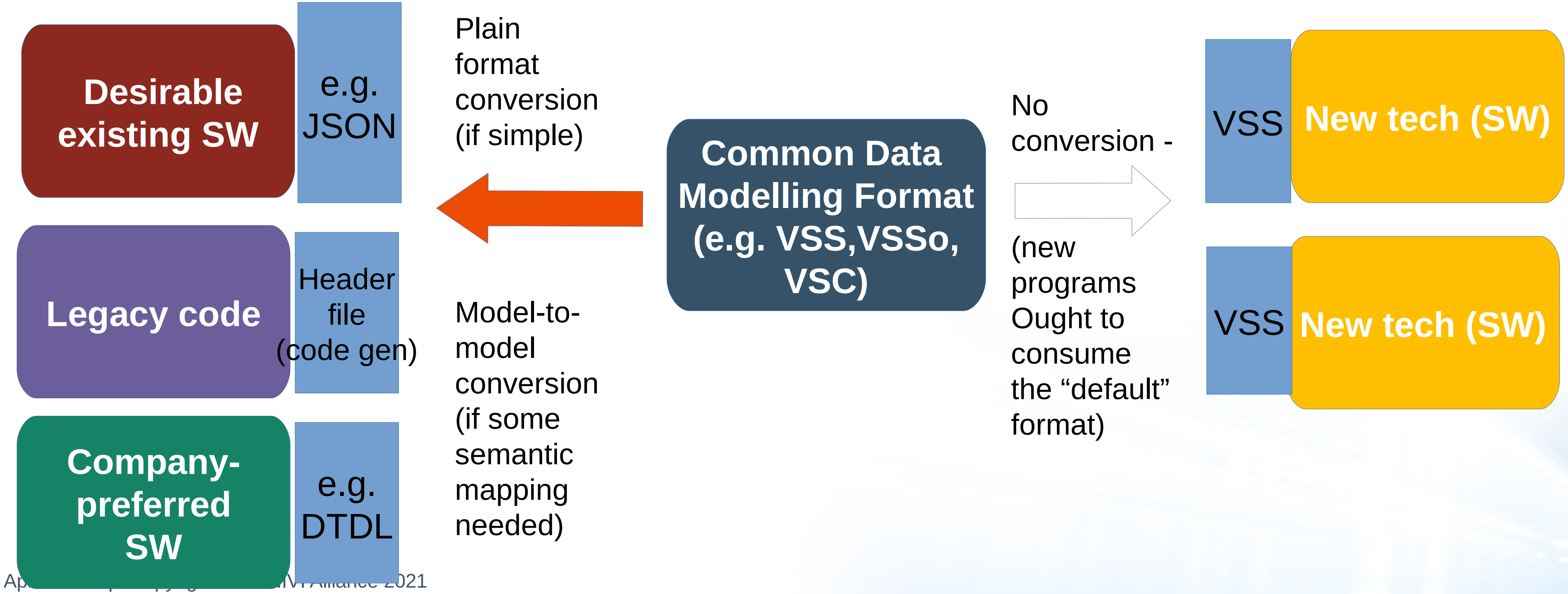
- **Cross-IDL portability**
  We need to be able to import (and export) existing IDL formats into a generic, easy-to-parse syntax while maintaining semantic equivalence

Data description format conversions

If there is a single model, when and why are conversions still needed?

Q: When and why do we convert from ***the common data model*** to to other formats?

A1: To interface with <u>existing</u> technology that *consumes* a different format! (e.g. DTDL for Microsoft solutions)
A2: Some strong driver for a certain choice (e.g. Web technology vastly favors JSON)
A3: Use a more advanced model-language (e.g. ontology) → additional information may be added from other source



Desirable existing SW

e.g. JSON

Legacy code

Header file (code gen)

Company-preferred SW

e.g. DTDL

Plain format conversion (if simple)

Model-to-model conversion (if some semantic mapping needed)

Common Data Modelling Format (e.g. VSS,VSSo, VSC)

No conversion -

(new programs Ought to consume the "default" format)

VSS

New tech (SW)

VSS

New tech (SW)

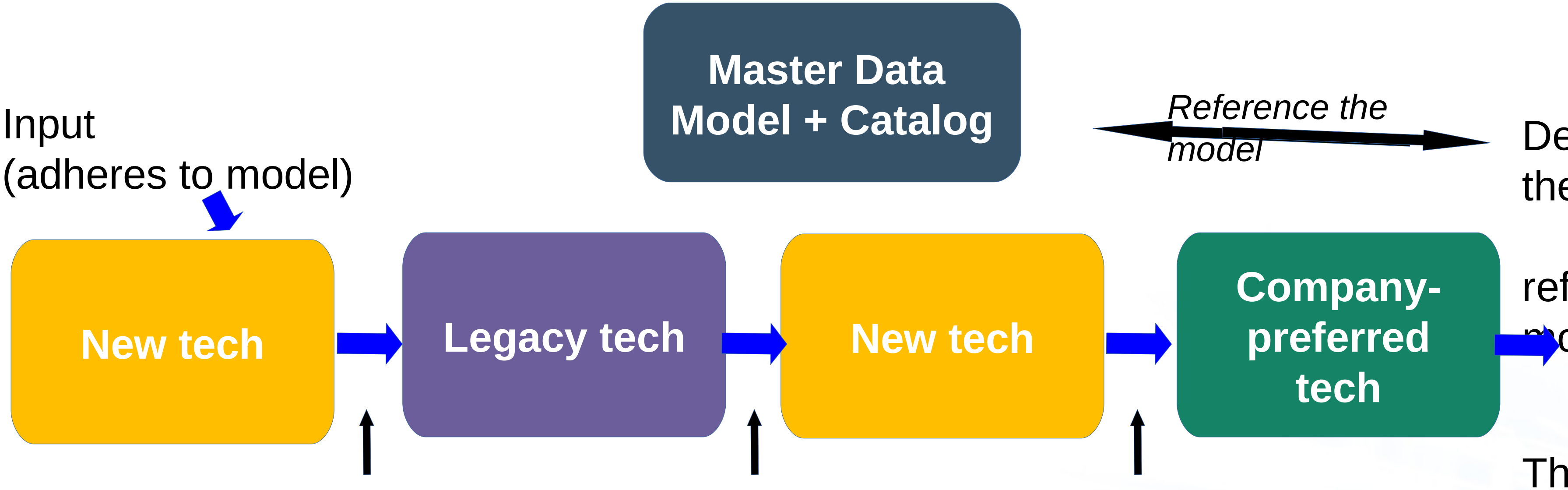
How is that conversion strategy different from today’s situation?

Q: Since we propose to sometimes convert to other formats/models, how is this different from the **conversion** of many different formats and models that is being done today?

A: Current conversions are simply **ad-hoc integrations** of many fragmented technologies without a plan.

Whereas, agreeing on one *central* data model means there is an expectation and agreement that the whole technology stack **maintains** that meaning / behavior / semantics, even in the presence of some conversions.

Meaning shall be maintained throughout any technology pipeline, even if made up of diverse technology. Fundamental things like the actual data name, datatype, unit is *uniquely defined* when there is one central model.



→ **Pipeline *might* contain conversions: Formats / protocols / data encodings, etc** →

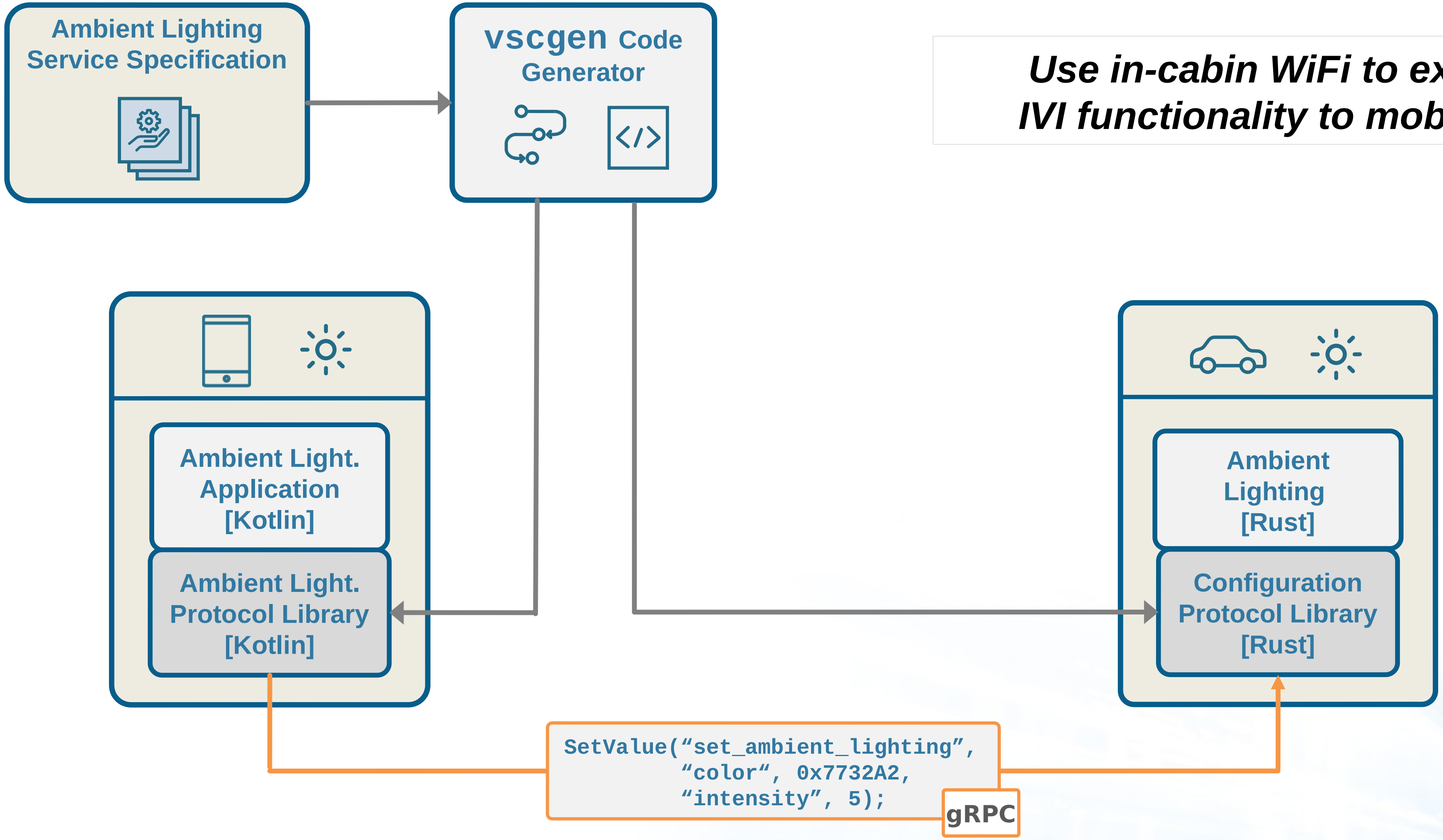
Despite diverse tech being used, the result can be ‘*understood*’ by referring back to one single model + catalog.

The output still adheres to the modelling: i.e. Name is unchanged, datatype remains, unit/meaning is as specified.

Usage Examples

BYOD Ambient Lighting Service Example

Use in-cabin WiFi to expose full IVI functionality to mobile device



Ambient Lighting Service Specification

vscgen Code Generator

Ambient Light. Application [Kotlin]

Ambient Light. Protocol Library [Kotlin]

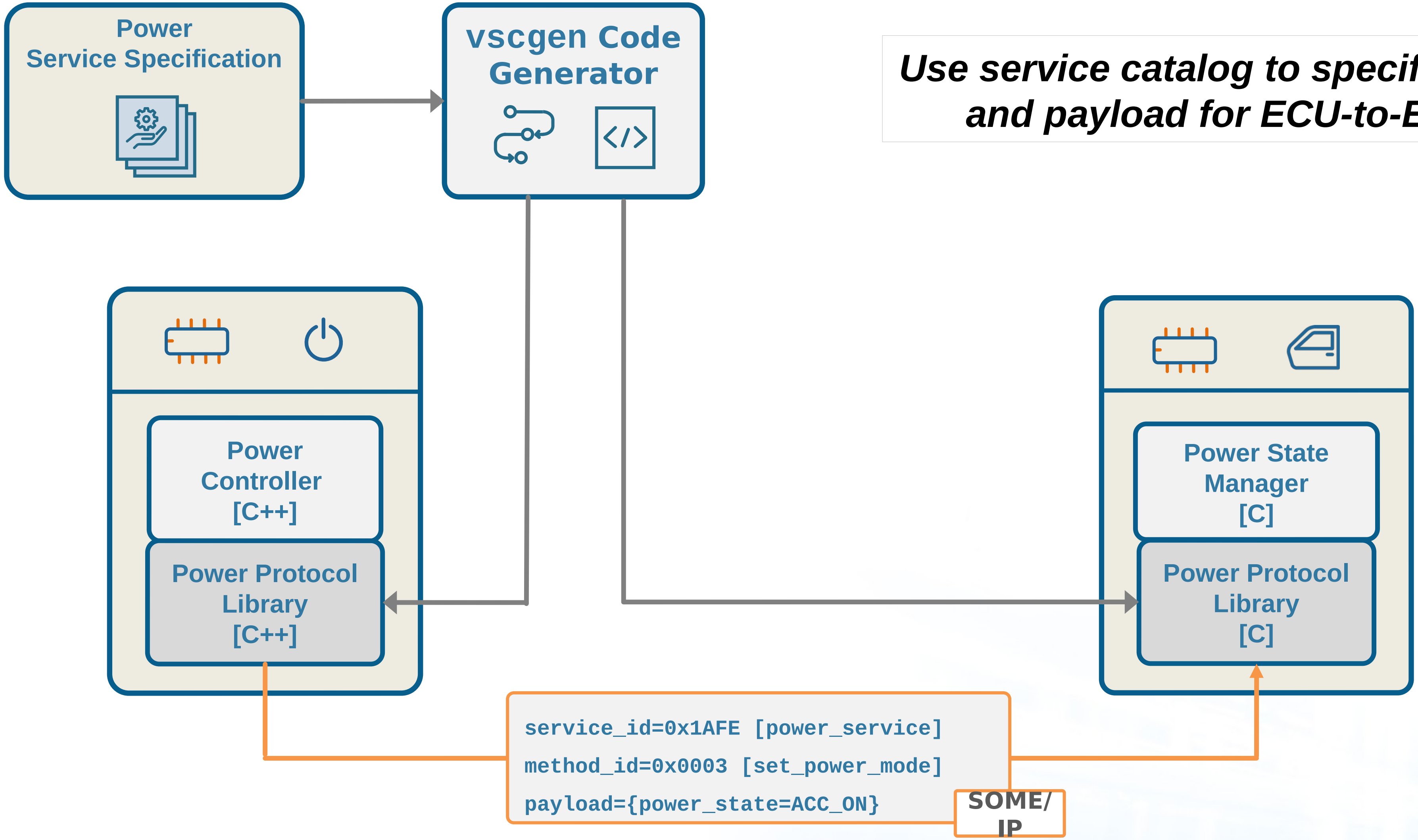
Ambient Lighting [Rust]

Configuration Protocol Library [Rust]

```
SetValue("set_ambient_lighting",
         "color", 0x7732A2,
         "intensity", 5);
```

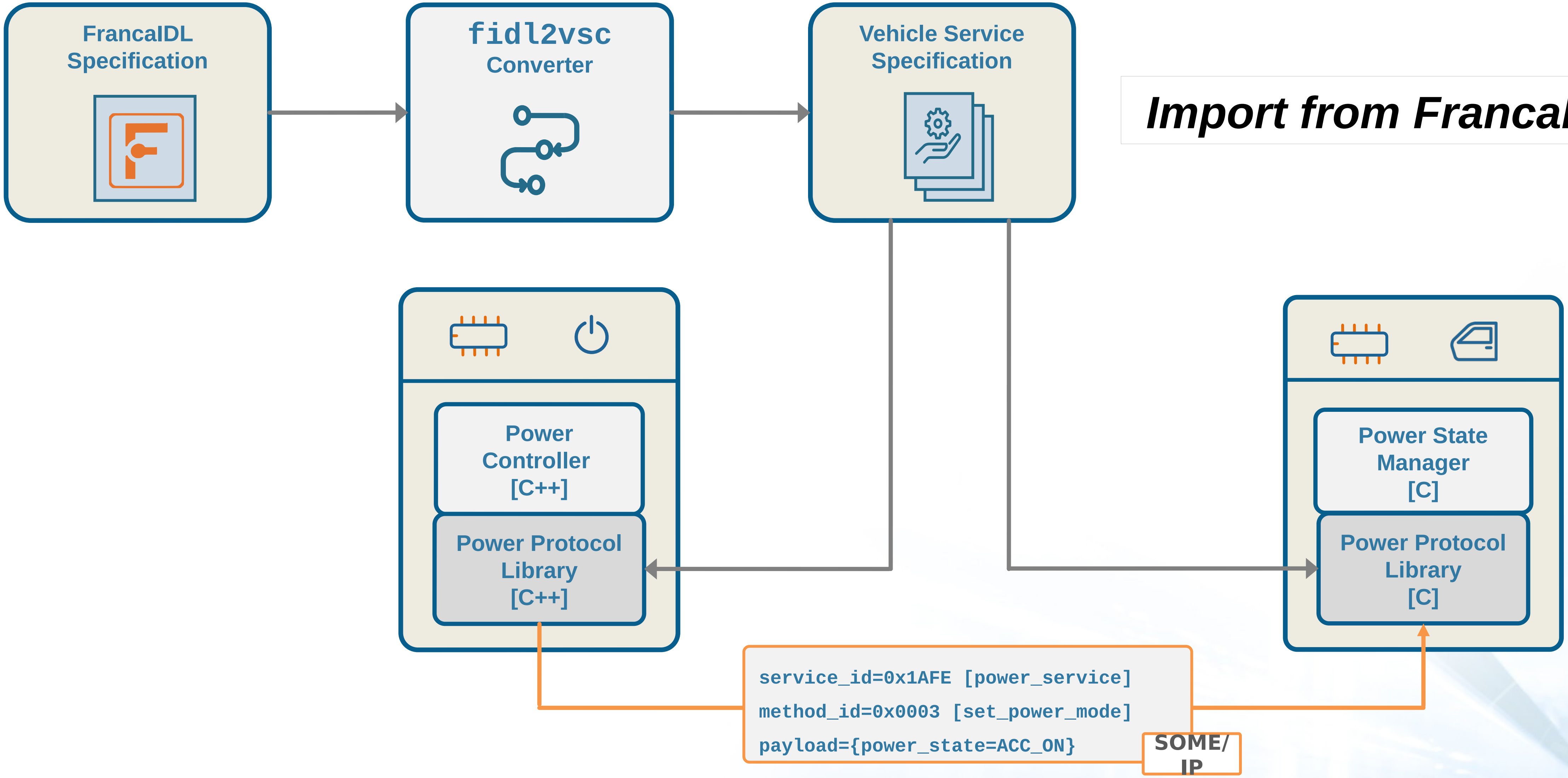
In-vehicle Network Example

Use service catalog to specify commands and payload for ECU-to-ECU traffic



```
service_id=0x1AFE [power_service]
method_id=0x0003 [set_power_mode]
payload={power_state=ACC_ON}
```

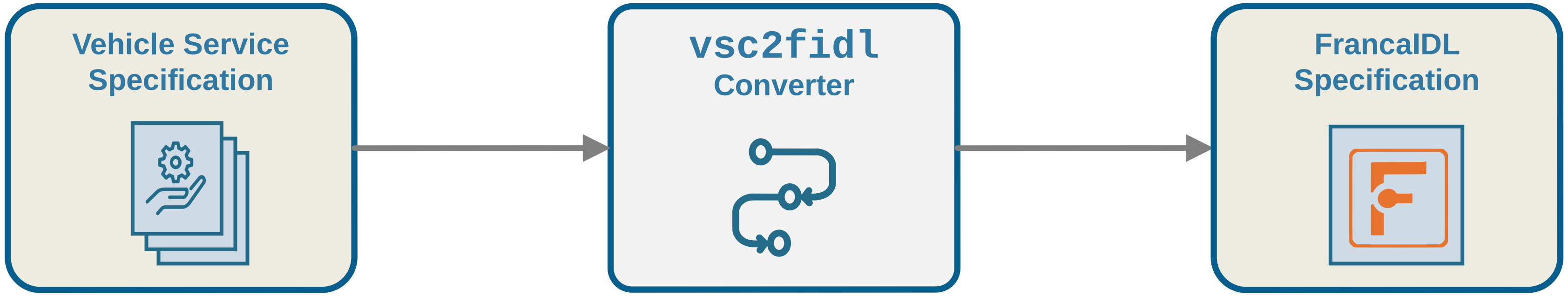
FrancaIDL Export Example



```
service_id=0x1AFE [power_service]
method_id=0x0003 [set_power_mode]
payload={power_state=ACC_ON}
```

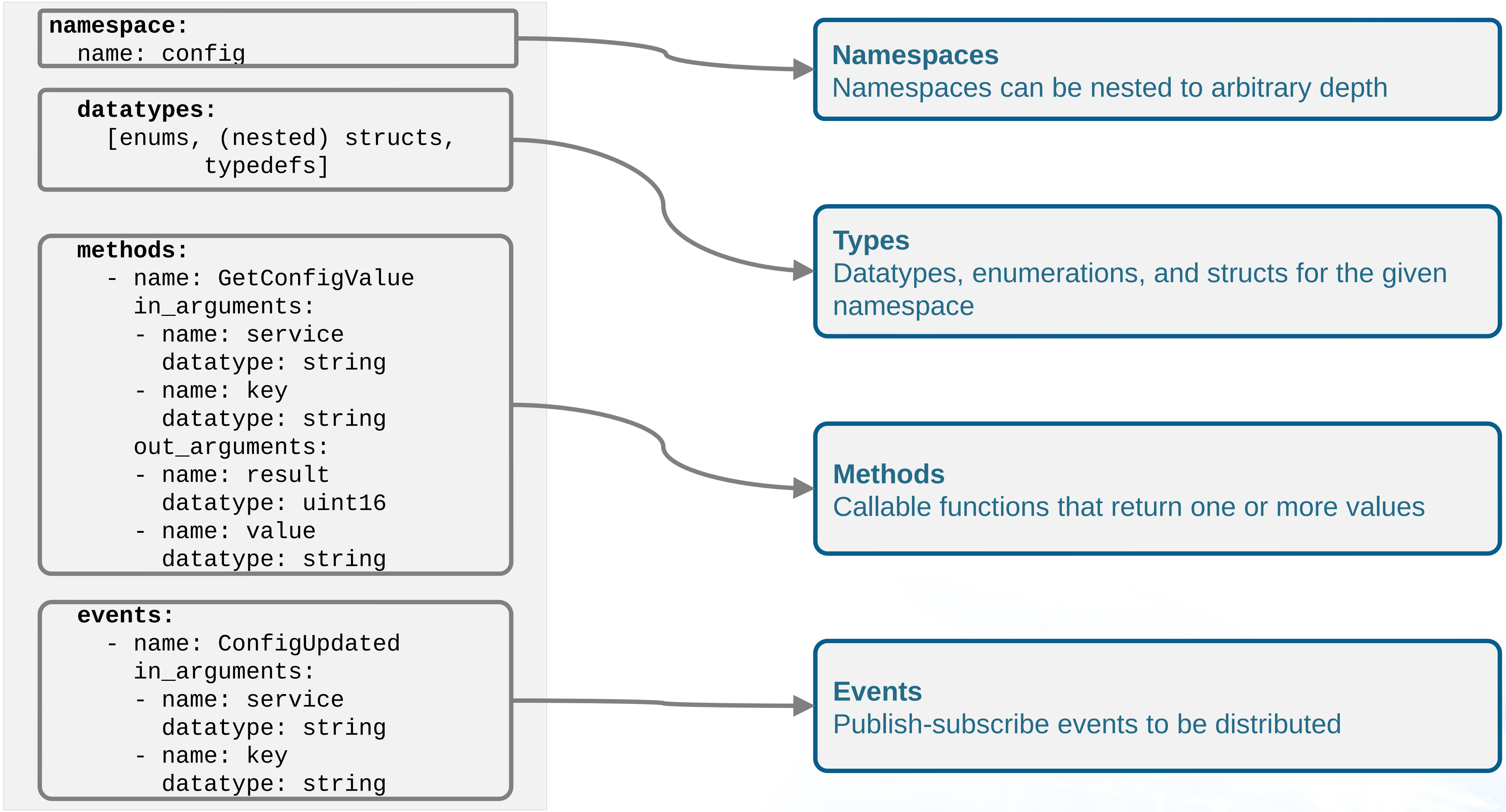
Import from FrancaIDL

FrancalDL Import Example



Export to FrancaIDL

Service Specification File Structure



```
namespace:
  name: config
```

```
  datatypes:
    [enums, (nested) structs,
           typedefs]
```

```
  methods:
    - name: GetConfigValue
      in_arguments:
      - name: service
        datatype: string
      - name: key
        datatype: string
      out_arguments:
      - name: result
        datatype: uint16
      - name: value
        datatype: string
```

```
  events:
    - name: ConfigUpdated
      in_arguments:
      - name: service
        datatype: string
      - name: key
        datatype: string
```

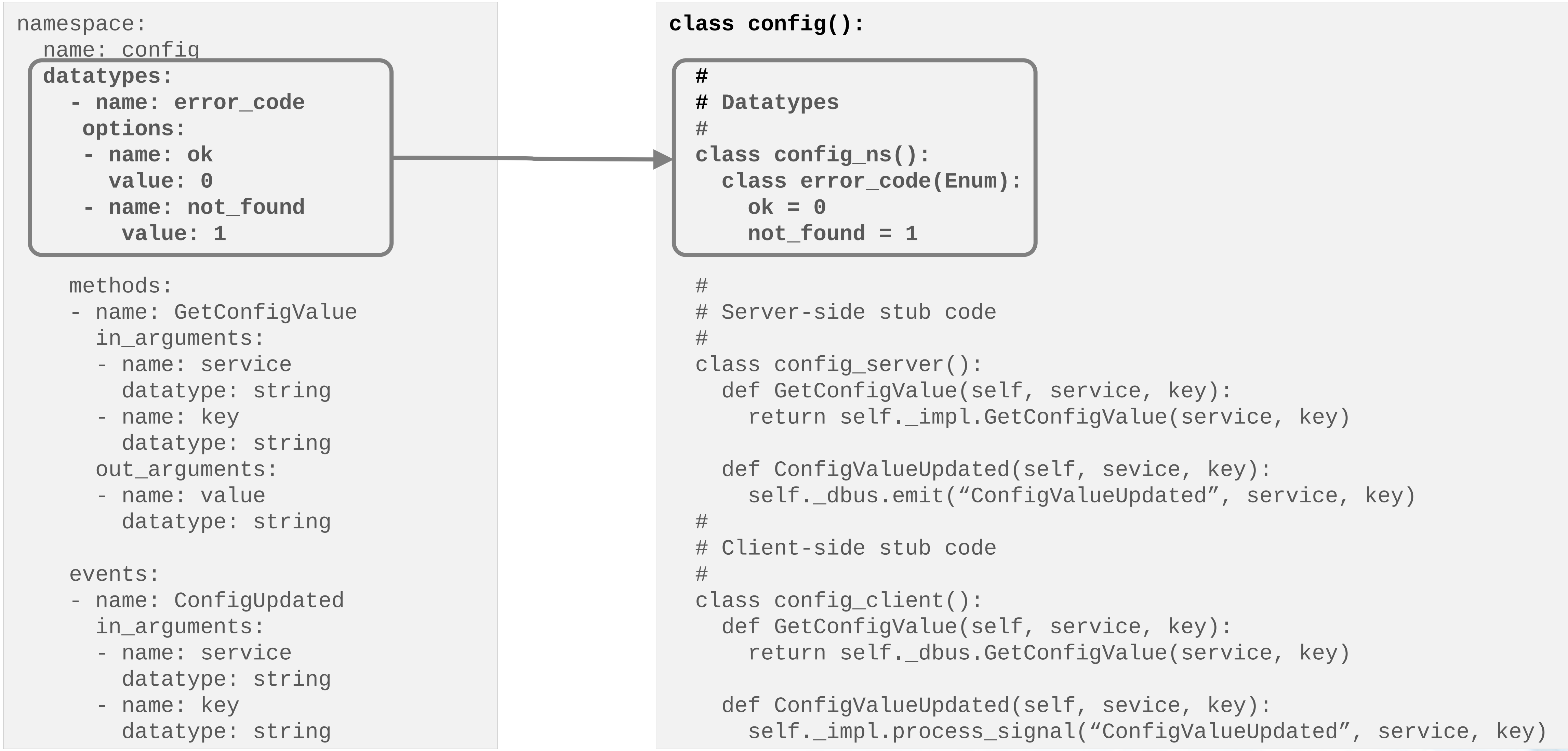
Namespaces
Namespaces can be nested to arbitrary depth

Types
Datatypes, enumerations, and structs for the given namespace

Methods
Callable functions that return one or more values

Events
Publish-subscribe events to be distributed

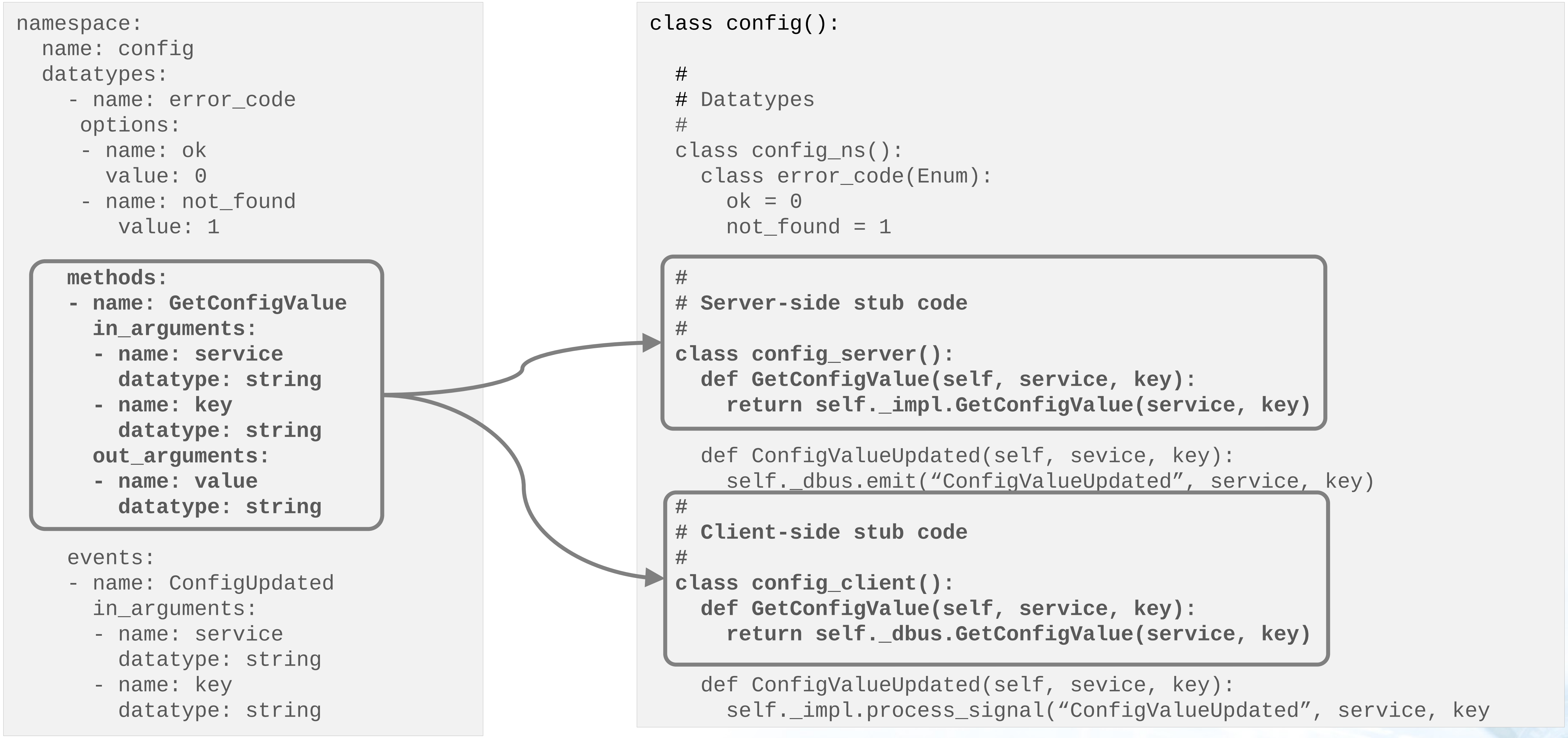
Service Specification → Datatypes



```
namespace:
 name: config
 datatypes:
  - name: error_code
   options:
   - name: ok
     value: 0
   - name: not_found
     value: 1

  methods:
  - name: GetConfigValue
    in_arguments:
    - name: service
      datatype: string
    - name: key
      datatype: string
    out_arguments:
    - name: value
      datatype: string

  events:
  - name: ConfigUpdated
    in_arguments:
    - name: service
      datatype: string
    - name: key
      datatype: string
```

```
class config():

 #
 # Datatypes
 #
 class config_ns():
   class error_code(Enum):
     ok = 0
     not_found = 1

 #
 # Server-side stub code
 #
 class config_server():
   def GetConfigValue(self, service, key):
     return self._impl.GetConfigValue(service, key)

   def ConfigValueUpdated(self, sevice, key):
     self._dbus.emit(“ConfigValueUpdated”, service, key)
 #
 # Client-side stub code
 #
 class config_client():
   def GetConfigValue(self, service, key):
     return self._dbus.GetConfigValue(service, key)

   def ConfigValueUpdated(self, sevice, key):
     self._impl.process_signal(“ConfigValueUpdated”, service, key)
```

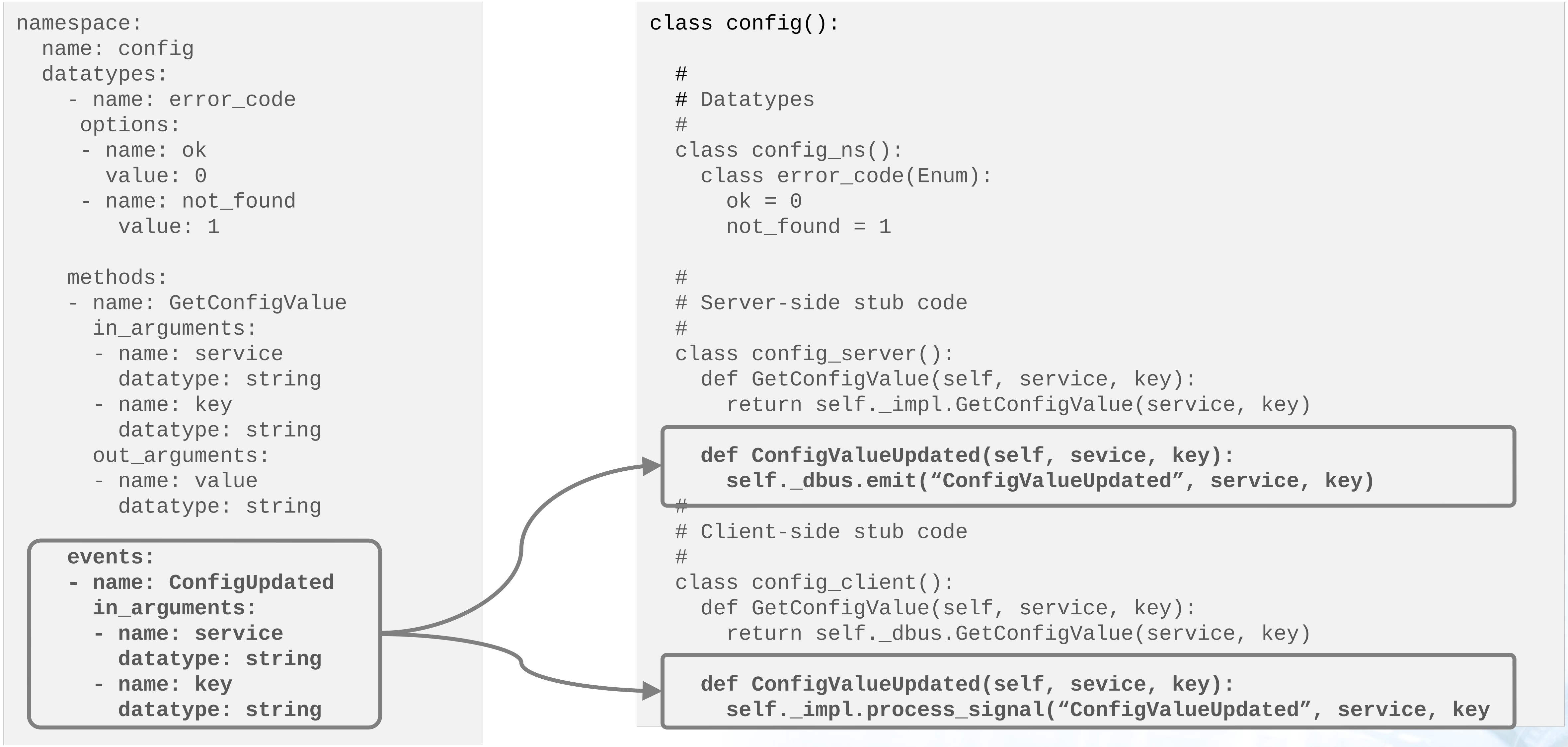
Service Specification → Methods



```
namespace:
  name: config
  datatypes:
    - name: error_code
     options:
     - name: ok
       value: 0
     - name: not_found
        value: 1

    methods:
    - name: GetConfigValue
      in_arguments:
      - name: service
        datatype: string
      - name: key
        datatype: string
      out_arguments:
      - name: value
        datatype: string

    events:
    - name: ConfigUpdated
      in_arguments:
      - name: service
        datatype: string
      - name: key
        datatype: string
```

```
class config():

  #
  # Datatypes
  #
  class config_ns():
    class error_code(Enum):
      ok = 0
      not_found = 1

  #
  # Server-side stub code
  #
  class config_server():
    def GetConfigValue(self, service, key):
      return self._impl.GetConfigValue(service, key)

    def ConfigValueUpdated(self, sevice, key):
      self._dbus.emit(“ConfigValueUpdated”, service, key)
  #
  # Client-side stub code
  #
  class config_client():
    def GetConfigValue(self, service, key):
      return self._dbus.GetConfigValue(service, key)

    def ConfigValueUpdated(self, sevice, key):
      self._impl.process_signal(“ConfigValueUpdated”, service, key
```

Service Specification → Events (pub/sub)



```
namespace:
  name: config
  datatypes:
    - name: error_code
     options:
     - name: ok
       value: 0
     - name: not_found
        value: 1

    methods:
    - name: GetConfigValue
      in_arguments:
      - name: service
        datatype: string
      - name: key
        datatype: string
      out_arguments:
      - name: value
        datatype: string

    events:
    - name: ConfigUpdated
      in_arguments:
      - name: service
        datatype: string
      - name: key
        datatype: string
```

```
class config():

  #
  # Datatypes
  #
  class config_ns():
    class error_code(Enum):
      ok = 0
      not_found = 1

  #
  # Server-side stub code
  #
  class config_server():
    def GetConfigValue(self, service, key):
      return self._impl.GetConfigValue(service, key)

    def ConfigValueUpdated(self, sevice, key):
      self._dbus.emit(“ConfigValueUpdated”, service, key)
  #
  # Client-side stub code
  #
  class config_client():
    def GetConfigValue(self, service, key):
      return self._dbus.GetConfigValue(service, key)

    def ConfigValueUpdated(self, sevice, key):
      self._impl.process_signal(“ConfigValueUpdated”, service, key
```

Nested Namespaces

```
namespaces:
- name: ivi
  namespaces:
  - name: fm-tuner
    methods:
    - name: SetFMFrequency
      in_arguments:
      - name: frequency
        datatype: float
        out_arguments:
      - name: result
        datatype: uint16
```

- Enables feature separation on interface level
- File imports can be done into specific namespace
- Arbitrary nesting depth
- Datatypes in namespaces can be addressed through absolute or relative paths

Deployment file structure

```
namespaces:
  - name: config
    dbus_interface: org.genivi.config
    methods:
    - name: GetConfigValue
      dbus_name: get-config-value
```

- Extends service specification file with language & protocol-specific information that must be known at build time
- Values used by code generator template
- Does not replace runtime configuration files

Examples:

- Name → Method ID mapping in SOME/IP
- Protocol conversions (little-endian, etc)
- Method and argument renaming to comply with language syntax (user-id → user_id)

Importing global definitions and interfaces

```
# global-errors.yml
namespace:
 name: global_error
   - datatypes:
   name: result
   datatype: enumeration
   options:
   - name: ok
     value: 0
```

```
namespace:
 name: configuration
  import:
  - file_name: global-errors.yml

  methods:
  - name: GetConfigValue
    out_argument: global_error.result
```

- Imports commands, methods, events, and datatypes
- Attached to the local namespace
- Generated code contains all imports
- Allows services to import globally defined interfaces that have to be implemented (life cycle management, diagnostics, etc)

Template files

```
## DBUS introspection XML file generation

<interface name='$iface.dbus_interface’>

#for $cmd in $iface.get('commands', [])
  <method name='$cmd.name’>

  #for $inarg in $cmd.get(in_arguments, [])
    <arg
      type='$dbus_support.
            convert_vsc_type_to_dbus($inarg)’
      name='$inarg.name’
      direction='in’
    />
  #end for

  </method>
#end for
```

- Uses Cheetah Python template library
- Each template generates code for specific language protocol stack combination
- Replaces tokens in template file with elements from service file parse tree
- Template for Rust/DBUS and Python/DBUS supported

Next steps

- **Open Source tooling**
  - MPLv2 / CC-BY-SA 4.0 licensing in progress
  - To be hosted by GENIVI

- **Agree on how we want to integrate Vehicle Signal Specification**

- **FrancaIDL integration**

- **Agree on W3C transport protocol as part of VSC/CVII technology stack**
  - Internet: gRPC, WAMP, JSON-RPC,
  - AUTOSAR integration: SOME/IP, DDS, ARA:COM, …

- **Create initial set of services**
  - Service proposals needed

The Common Vehicle Interface *Initiative*

De-fragmenting the industry is possible

This was just a taste…

- Refer to our references (a lot more details)
- Join active projects and upcoming workshops
- Contact us for more discussion or to get involved!

Coming up: Today’s Q&A and discussion.

References: Start here (https://at.projects.genivi.org/wiki/x/n4DNAw)
or go to **projects.genivi.org** and search for “Common Vehicle Interface...” (home page)

Thank You!

Contact the speakers:
magnus@feuerworks.com
gandersson@genivi.org

Contact W3C Transport and Automotive groups:

ted@w3.org

Visit GENIVI:

http://www.genivi.org

http://projects.genivi.org