GENIVI®

GENIVI Development Platform Activities

October 10, 2018 | GENIVI Technical Summit, Bangalore

Gunnar Andersson

Development Lead, GENIVI Alliance

Code Development Overview

GENIVI Code development

- Mission: Set automotive standards & specifications and produce code maintained within or by the GENIVI Alliance
  - GENIVI Development Platform
  - Software Components
  - GENIVI Baseline
  - Domain-interaction technologies
  - Software Development Environment

GENIVI Code development

- Supporting activities and functions
  - Confluence & JIRA
  - Continuous Integration/Deployment
  - Quality & Maintenance Policy
  - GENIVI Open Source Policy and License Review Team

- Google Summer of Code project
  - Machine learning and voice control

GENIVI Development Platform (GDP)

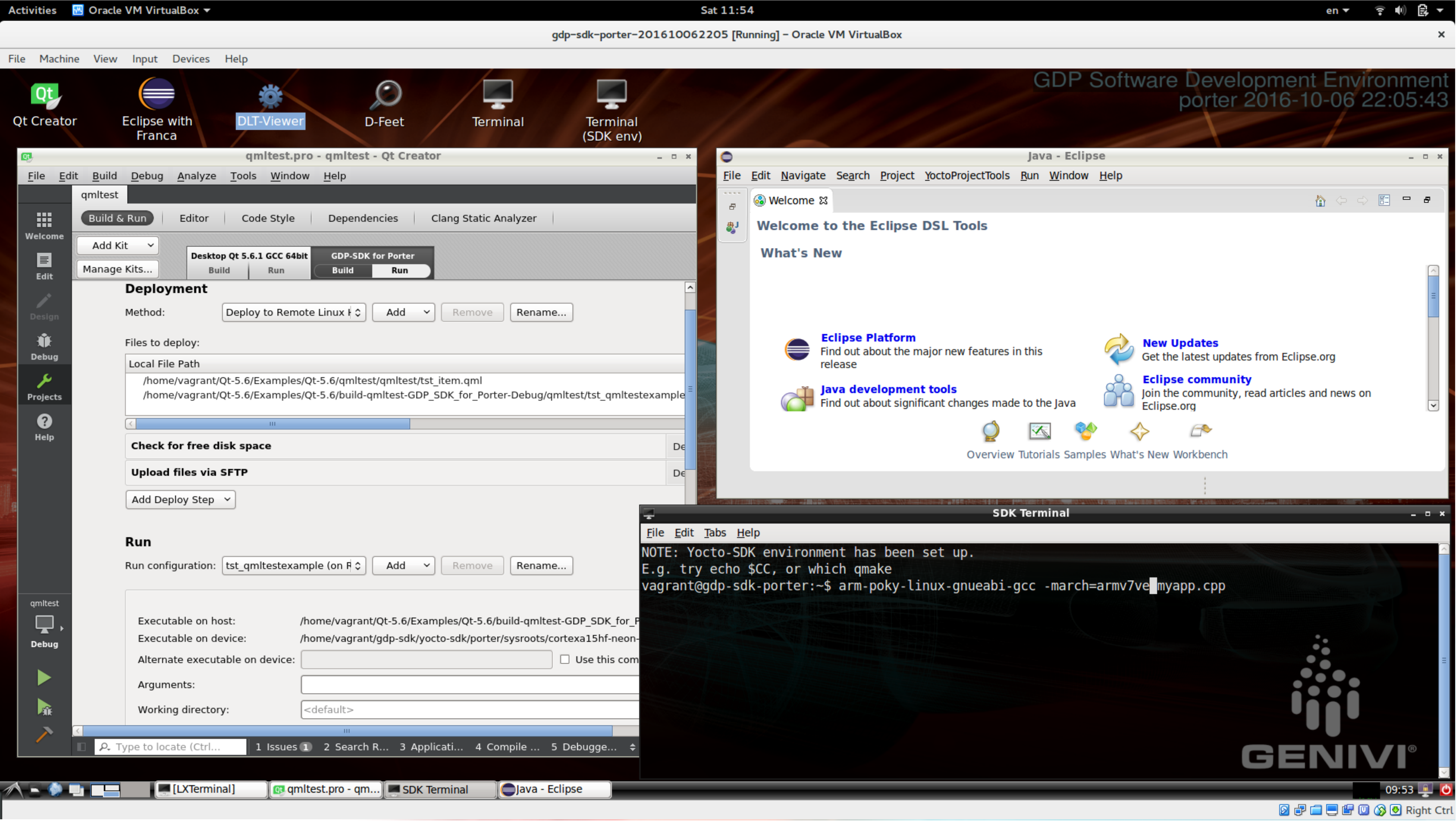
- Yocto build system
- Normally 2 Released/tagged major versions per year (but we recommend “GDP Master” as a rolling release)
- Proving ground for GENIVI software
- And starting point for a lot of product development
- “IVI demo” → changed to **development & platform focus**

GENIVI Development Platform (GDP)

- GDP is a...
- … Yocto-built,
- … Continuous-Integration-backed,
- … Profile divided,
- … Simple,
- … Logical,
- Base platform for early development of Automotive Software
- … with integrated GENIVI technologies, + matching SDE
- … and trying out useful “non-GENIVI” tech (e.g. NiFi Big Data)

GENIVI Development Platform – Recent Updates

- OSTree support (some targets)
- Lots of recipe/project quality-improvements, cleanups
- Updates to latest GENIVI baseline
- Updated BSPs
- Updated to support **rocko** and **sumo** branches, with **thud** to follow shortly.
- Note: Easy python dev (see new web page)
- "Big Data" support: MiNiFi C++
- Profiles



Deployment

Method: Deploy to Remote Linux H | Add | Remove | Rename...

Files to deploy:

Local File Path
/home/vagrant/Qt-5.6/Examples/Qt-5.6/qmltest/qmltest/tst_item.qml
/home/vagrant/Qt-5.6/Examples/Qt-5.6/build-qmltest-GDP_SDK_for_Porter-Debug/qmltest/tst_qmltestexample

Check for free disk space

Upload files via SFTP

Add Deploy Step

Run

Run configuration: tst_qmltestexample (on R | Add | Remove | Rename...

Executable on host: /home/vagrant/Qt-5.6/Examples/Qt-5.6/build-qmltest-GDP_SDK_for_P

Executable on device: /home/vagrant/gdp-sdk/yocto-sdk/porter/sysroots/cortexa15hf-neon-

Alternate executable on device: ☐ Use this com

Arguments:

Working directory: <default>

Type to locate (Ctrl... 1 Issues 1 2 Search R... 3 Applicati... 4 Compile ... 5 Debugge...

Welcome to the Eclipse DSL Tools

What's New

Eclipse Platform
Find out about the major new features in this release

Java development tools
Find out about significant changes made to the Java

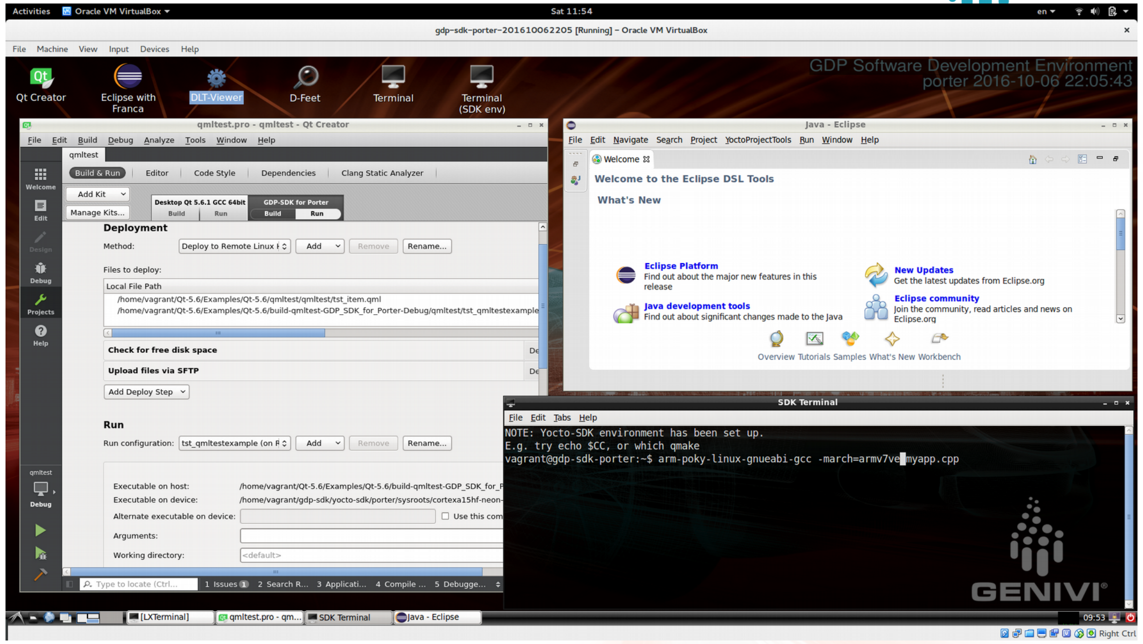
New Updates
Get the latest updates from Eclipse.org

Eclipse community
Join the community, read articles and news on Eclipse.org

```
NOTE: Yocto-SDK environment has been set up.
E.g. try echo $CC, or which qmake
vagrant@gdp-sdk-porter:~$ arm-poky-linux-gnueabi-gcc -march=armv7ve myapp.cpp
```

Software Development Environment (SDE)

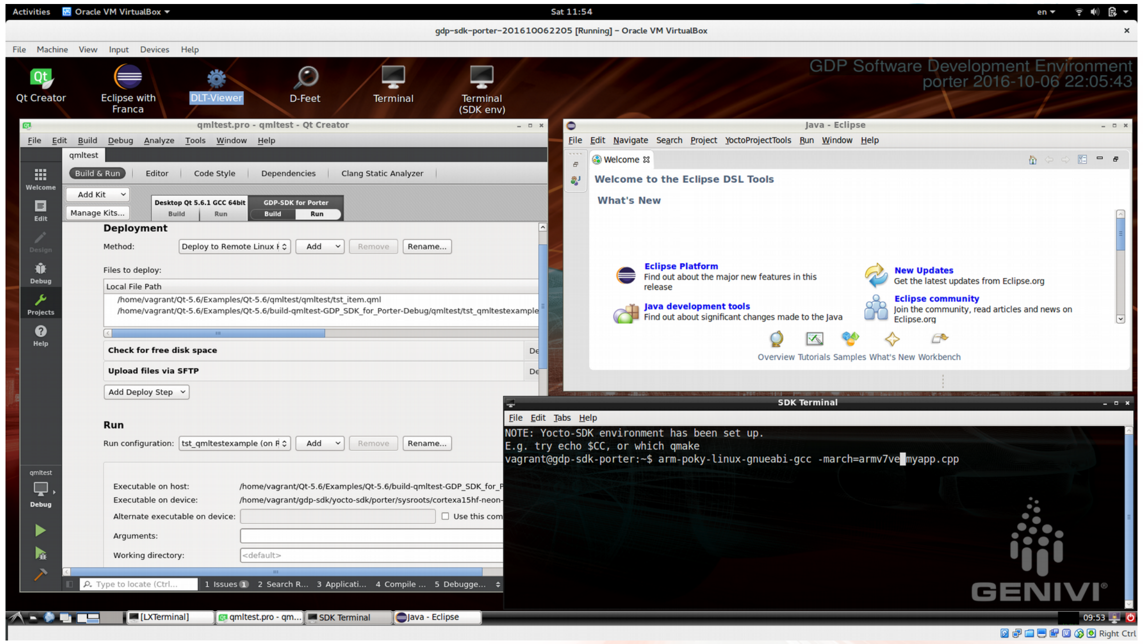
- Providing a integrated starting point for development:
- Matching Yocto SDK
- GENIVI tools
  - DLT Viewer
  - Franca/Common-API code-generators
- Qt Creator with ready-to-deploy settings
- Eclipse with Franca & plugins installed


gdp-sdk-porter-201610062205 [Running] - Oracle VM VirtualBox
GDP Software Development Environment
porter 2016-10-06 22:05:43
Qt Creator
Eclipse with Franca
DLT-Viewer
D-Feet
Terminal
Terminal (SDK env)
qmltest.pro - qmltest - Qt Creator
Deployment
Deploy to Remote Linux
Check for free disk space
Upload files via SFTP
Add Deploy Step
Run
Java - Eclipse
Welcome to the Eclipse DSL Tools
What's New
Eclipse Platform
Java development tools
New Updates
Eclipse community
Overview Tutorials Samples What's New Workbench
SDK Terminal
GENIVI

```
NOTE: Yocto-SDK environment has been set up.
E.g. try echo $CC, or which qmake
vagrant@gdp-sdk-porter:~$ arm-poky-linux-gnueabi-gcc -march=armv7ve myapp.cpp
```

Software Development Environment (SDE)

- ...being fundamentally reworked
- for flexibility, quicker build & updates.
- Lots of ideas. Less time.


gdp-sdk-porter-201610062205 [Running] - Oracle VM VirtualBox
Qt Creator
Eclipse with Franca
DLT-Viewer
D-Feet
Terminal
Terminal (SDK env)
GDP Software Development Environment
porter 2016-10-06 22:05:43
qmltest.pro - qmltest - Qt Creator
Java - Eclipse
Welcome to the Eclipse DSL Tools
SDK Terminal
GENIVI

Deployment

Method: Deploy to Remote Linux

Files to deploy:

Local File Path
/home/vagrant/Qt-5.6/Examples/Qt-5.6/qmltest/qmltest/tst_item.qml
/home/vagrant/Qt-5.6/Examples/Qt-5.6/build-qmltest-GDP_SDK_for_Porter-Debug/qmltest/tst_qmltestexample

Check for free disk space

Upload files via SFTP

Run

Run configuration: tst_qmltestexample (on R

Executable on host: /home/vagrant/Qt-5.6/Examples/Qt-5.6/build-qmltest-GDP_SDK_for_F
Executable on device: /home/vagrant/gdp-sdk/yocto-sdk/porter/sysroots/cortexa15hf-neon-
Alternate executable on device:
Arguments:
Working directory: <default>

```
NOTE: Yocto-SDK environment has been set up.
E.g. try echo $CC, or which qmake
vagrant@gdp-sdk-porter:~$ arm-poky-linux-gnueabi-gcc -march=armv7ve myapp.cpp
```

- As always, show interest & offer to help and the priority will be increased

Releases?

GDP 14.0 (rocko-based) is tagged and ready!

GDP 14.1 (sumo-based) is tagged and ready!

As always, GDP Master (plus frequent communication *among* developers) is recommended.

GENIVI Development Platform – Variants & Profiles

- Starting with GDP 14 a long-expected “feature”

- GDP is split into 3 variants (a.k.a. profiles)
  - **Core :** A core to build on. No graphics.
  - **IVI** : Everything (almost). Also, “compliant”
  - **Data :** cloud-connectivity, telematics & “big data”

GENIVI Development Platform - potentials

- Of course, profiles build-out
- Flatpak build-out
  - Rock-solid containment (RedHat standard tech)
  - Tools & standard method for application dev already there
  - Versioned, fully controlled application build environment
  - Versioned, fully controlled runtime
  - Storage & update in OSTree, (just like the system)
- GDP partial impl. (container sandbox) since v12. Application “handling” still pending interest
- Various ideas around great Yocto CI/CD speed-up and SDE

GENIVI Development Platform – support functions

- **Go** or **GoCD** nowadays (https://go.cd)
- We needed pipeline focus from the start
- Somewhat similar to Jenkins plus various plugins
- GDP CI/CD is done at https://go.genivi.org
- Details, instructions, account see GENIVI Wiki
- GENIVI “CI Policy”: **Go** for **GDP**, GitHub integrated services for components (Travis, AppVeyor, Semaphore, CircleCI...)

GENIVI Development Platform – other support functions

- Public Confluence and JIRA
- Email to: genivi-projects@lists.genivi.org

GENIVI Development Platform – CIAT infrastructure

- Pipelines because:
  - Logically separate stages   Build, Test, Upload
  - Chain pipelines together
  - Control of artifacts going into and out of each stage
- Go.cd because:
  - Pipelines from the beginning. We knew we needed them
  - Powerful templating with combination of Parameters & Env. Variables
  - It was familiar (for the person installing it)
  - One-shot install, less plugins
- There are other systems... potentially even better

Quality & Maintenance Policy – Scope

- The document outlines *code quality* requirements and *maintenance* processes for all* software hosted **on GENIVI GitHub account**.
- There are different requirements depending on maintenance level
- These are minimum common principles only – each maintainer retains a lot of freedom to define the details (see later slides), and can of course do *more*.

**Except if GENIVI GitHub hosts a fork which has a clear upstream location defining its own processes.*

Background & Needs

- Promote trust in GENIVI code quality and development processes
- Quickly understand which software projects are still maintained
- Document process/policy to back up our interaction with maintainers
- Agree on *expectations* between GENIVI and code maintainers for *all* projects hosted on GENIVI's GitHub account
- Accountability to a mutually agreed standard & process for recognizing and addressing any project issues with maintainer (manager).
- *Provide better information* to our community, what to *expect,* and how to resolve issues

Component Maintenance Level

1. **Maintenance level** **Expectations**

ACTIVE	**Response time**: 1-3 days **Resolution time**: ~1 week
PARTIAL	**Response time**: 1-2 weeks **Resolution time**: ~2-4 weeks
NOT ACTIVE	**Response time**: N/A **Resolution time**: N/A

Response time: Reading and submitting a first on-topic response to Tickets/Issues, Mailing List questions, and submitted patches / pull-requests.

Resolution time: Discussing towards, and reaching, a conclusion to close a ticket, merge (or reject) a patch or pull-request. (Understandably, resolution time can differ depending on the situation).

Quality & Maintenance

- All GENIVI GitHub projects shall follow the Q&M policy

- Requirements and guidelines around these topics:

  - Continuous Build
  - Community communication
  - Static Analysis
  - Automated Testing
  - Commit messages, etc.

Google Summer of Code Project

Video presentation

Including the slides at the end of this presentation, plus narration.

Also including live demo of the system.

Video linked from this page : https://at.projects.genivi.org/wiki/x/xImxAQ

GSoC slides

Here follows a copy of GSoC project slides.
(They were showed on the video but are here so we can refer to them)

GENIVI - Google Summer of Code

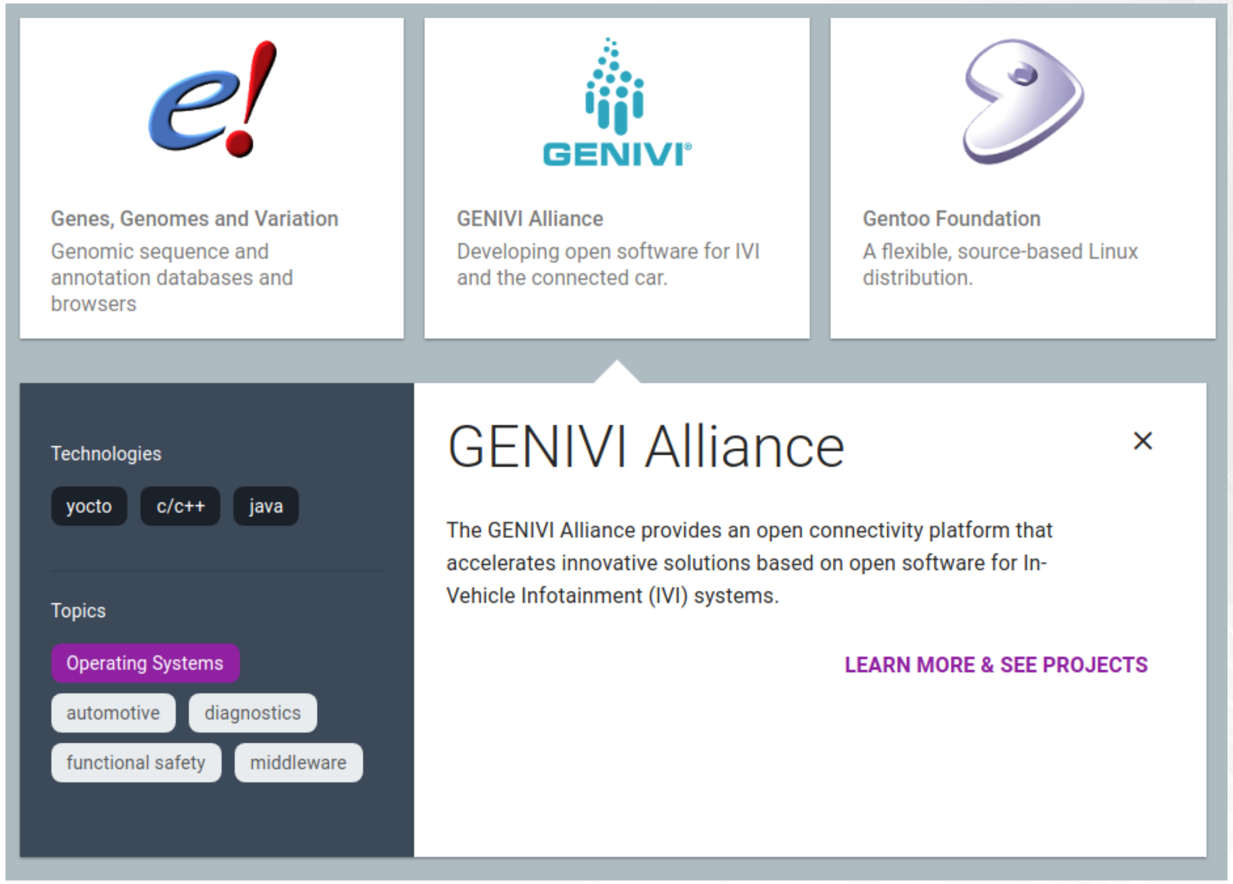
2018 project

Chandeepa Dissanayake

Gunnar Andersson, GENIVI

Jeremiah Foster, Luxoft

GENIVI's 2018 Google Summer of Code project



Genes, Genomes and Variation
Genomic sequence and annotation databases and browsers

GENIVI Alliance
Developing open software for IVI and the connected car.

Gentoo Foundation
A flexible, source-based Linux distribution.

Technologies
yocto c/c++ java

Topics
Operating Systems
automotive diagnostics
functional safety middleware

GENIVI Alliance

The GENIVI Alliance provides an open connectivity platform that accelerates innovative solutions based on open software for In-Vehicle Infotainment (IVI) systems.

LEARN MORE & SEE PROJECTS

- ❑ GENIVI participated for the second year in the Google sponsored "Summer of Code".
- ❑ Google Summer of Code is a global program focused on introducing students to open source software development. Students work on a 3 month programming project with an open source organization during their break from university.
- ❑ Project focus this year was on using voice commands to an InVehicle Infotainment system. This ambitious project integrated state of the art technologies and had a successful outcome thanks to the student and mentors.

GSoC results

The GSoC project

Provides a stipend to a student so they can focus on code

Teaches a student about key interactions in developing open source software

Ideally it provides a new contributor to an open source project

Code, proof-of-concepts, bug fixes, documentation and other artifacts are provided to the community at large which can be reused

Results and artifacts

- Integration of a Text-To-Speech (TTS) solution on a GENIVI target
- Test of an end to end TTS and Voice Control integrated system with Google Cloud
- On hardware and Qemu emulated testing
- Recipes and strategy for embedded targets built with Yocto

Participants

- **Chandeepa Dissanayake** -- Participating student. Responsible for input on specification as well as how the resulting implementation meets the specification. Responsible for technology selection, discussion of technology selection with mentors, implementation of technology including any additional source code.

- **Gunnar Andersson** -- GENIVI lead developer. Played key technical mentorship role, responsible for advising the student on technical choices as it relates to GENIVI. Gave guidance in implementation as well as technical assistance on integration in GENIVI's development platform. Code and technology review.

- **Jeremiah Foster** -- GENIVI community manager. Project management, liaison with GSoC org, documentation and dissemination.

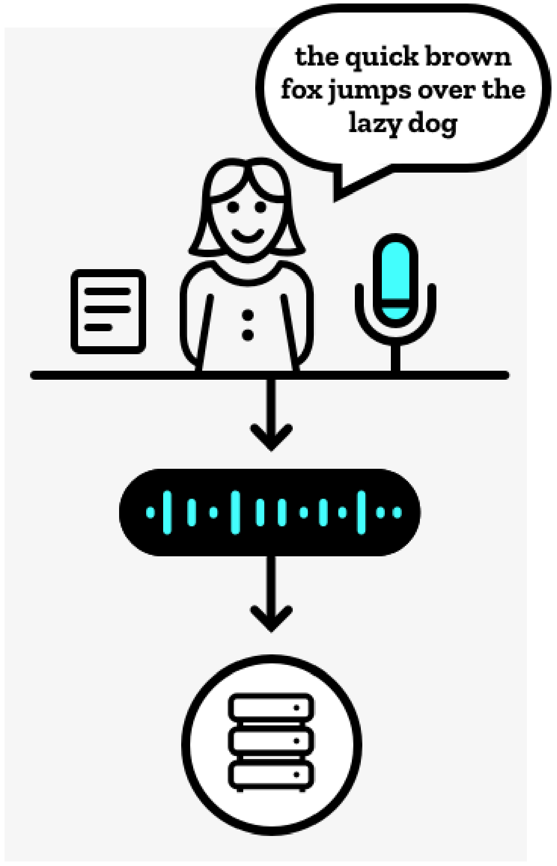
Functionality

- At a high level Voice Control for IVI Next Generation (VCIVING) implements voice commands on GENIVI’s InVehicle Infotainment platform.

- It provides mechanism for
  - capturing speech input via a microphone
  - transform speech into text
  - text recognition via a neural network
  - processing the resulting commands

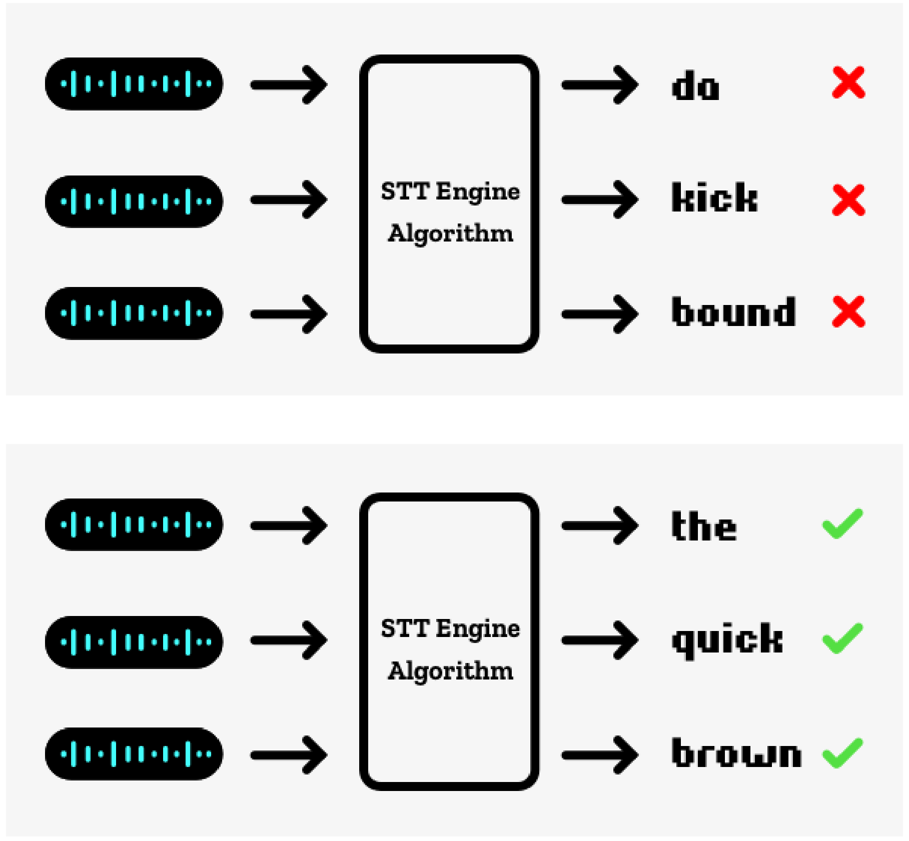
How a Speech Application Learns

Step 1. **Record voices**



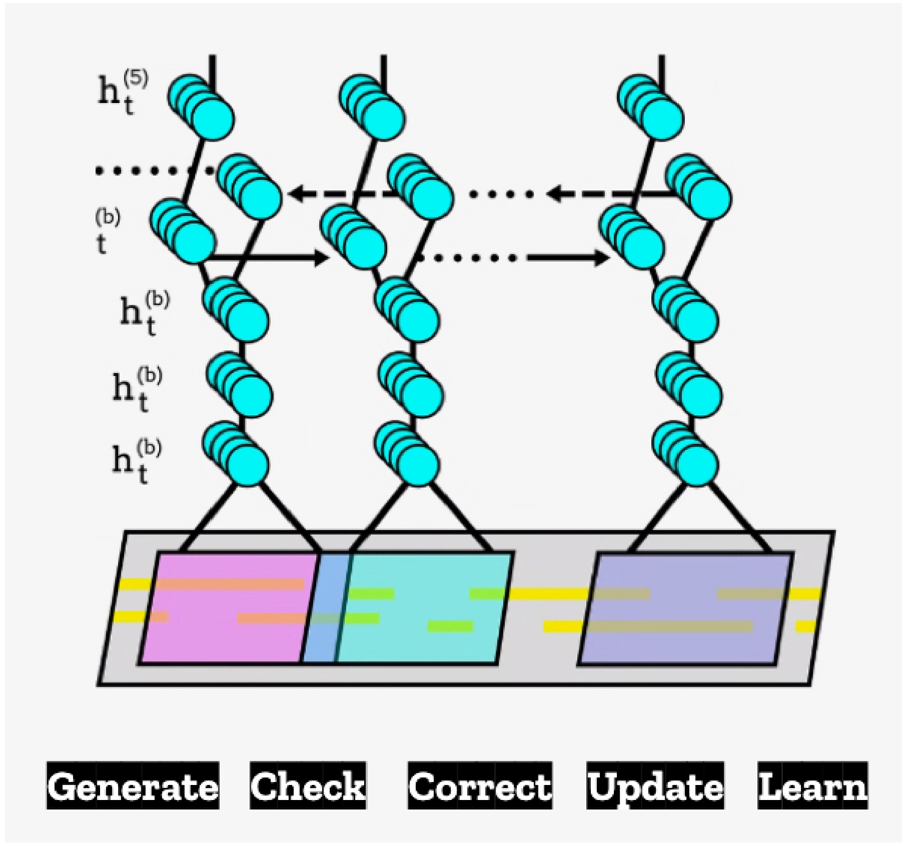
Common Voice Project

Step 2. **Input voice data**



Open Source STT Engine

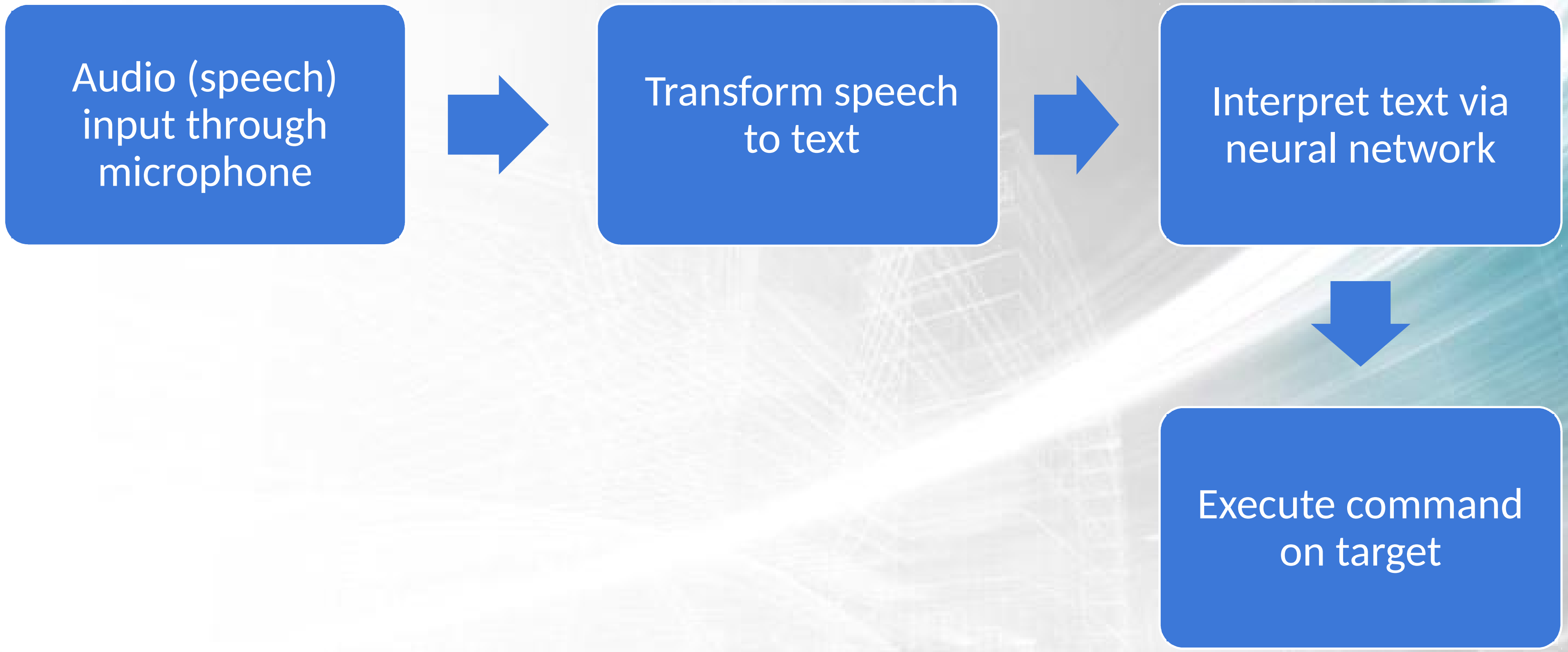
Step 3. **Train the speech algorithm**



Deep Learning Architecture

VCIVING Pipeline

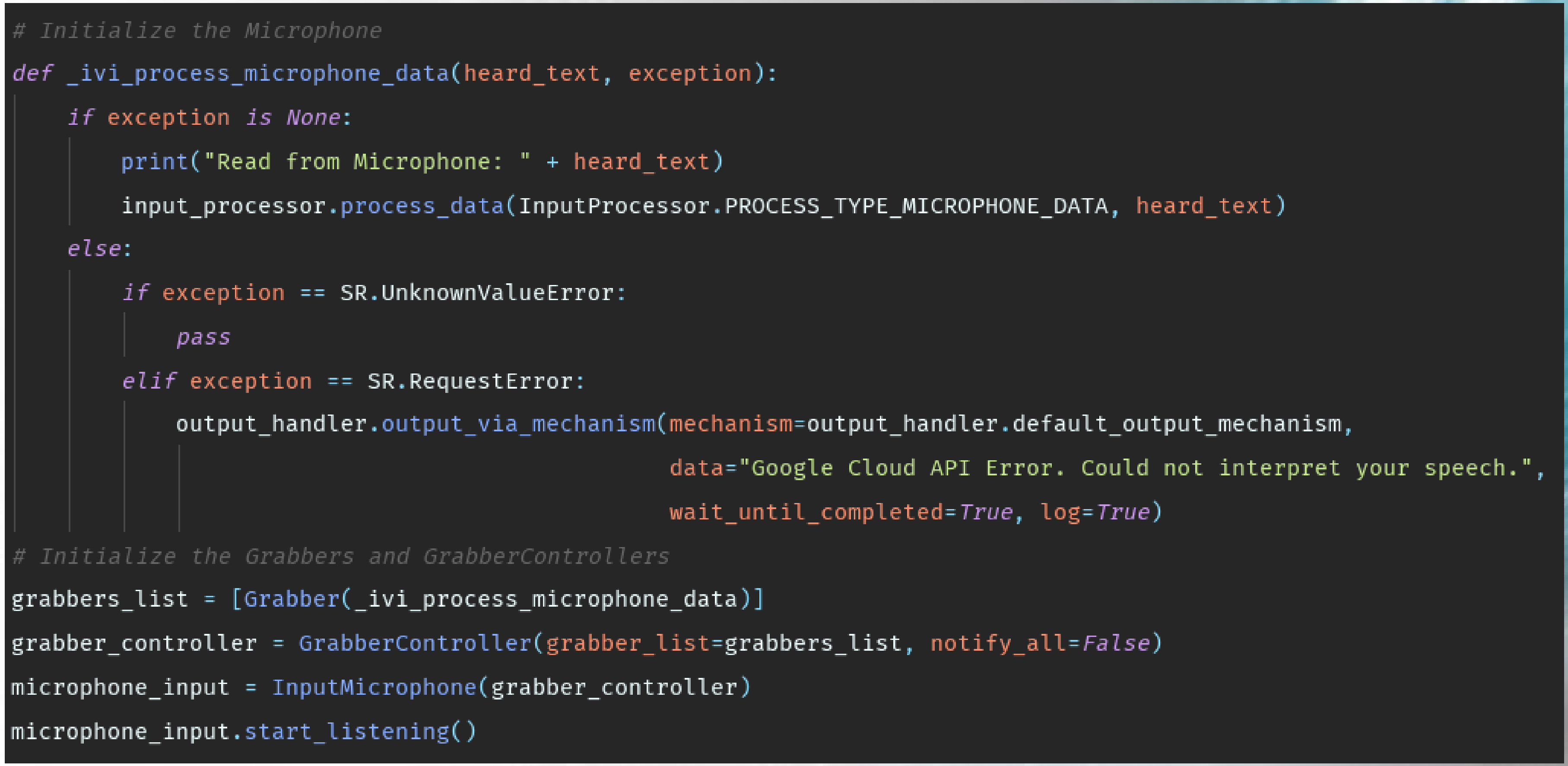
- VCIVING pipeline is composed of four consecutive steps


Audio (speech) input through microphone
Transform speech to text
Interpret text via neural network
Execute command on target

GENIVI®

VCIVING Pipeline Step 1: Capturing Input

Just as in the Mozilla example, the first step is to capture speech input from the user. The python code used uses a microphone as the input mechanism, then passes that to the speech recognition functions.



```
# Initialize the Microphone
def _ivi_process_microphone_data(heard_text, exception):
    if exception is None:
        print("Read from Microphone: " + heard_text)
        input_processor.process_data(InputProcessor.PROCESS_TYPE_MICROPHONE_DATA, heard_text)
    else:
        if exception == SR.UnknownValueError:
            pass
        elif exception == SR.RequestError:
            output_handler.output_via_mechanism(mechanism=output_handler.default_output_mechanism,
                                                data="Google Cloud API Error. Could not interpret your speech.",
                                                wait_until_completed=True, log=True)
# Initialize the Grabbers and GrabberControllers
grabbers_list = [Grabber(_ivi_process_microphone_data)]
grabber_controller = GrabberController(grabber_list=grabbers_list, notify_all=False)
microphone_input = InputMicrophone(grabber_controller)
microphone_input.start_listening()
```

Input: User’s speech through mic

Output: Digital audio of user’s voice

GENIVI®

VCIVING Pipeline Step 2: Speech-to-Text

- COTS hardware plugged directly into a USB port on a Raspberry Pi 3 is sufficient for voice capture on GNU/Linux
- Once we have the audio data, we convert it into text for processing

- Adopting any technology such as pre-trained model or third party API, the audio data is converted into text. (Implementation dependent)
- Google's Speech To Text API then returns back the textual representation of the captured speech

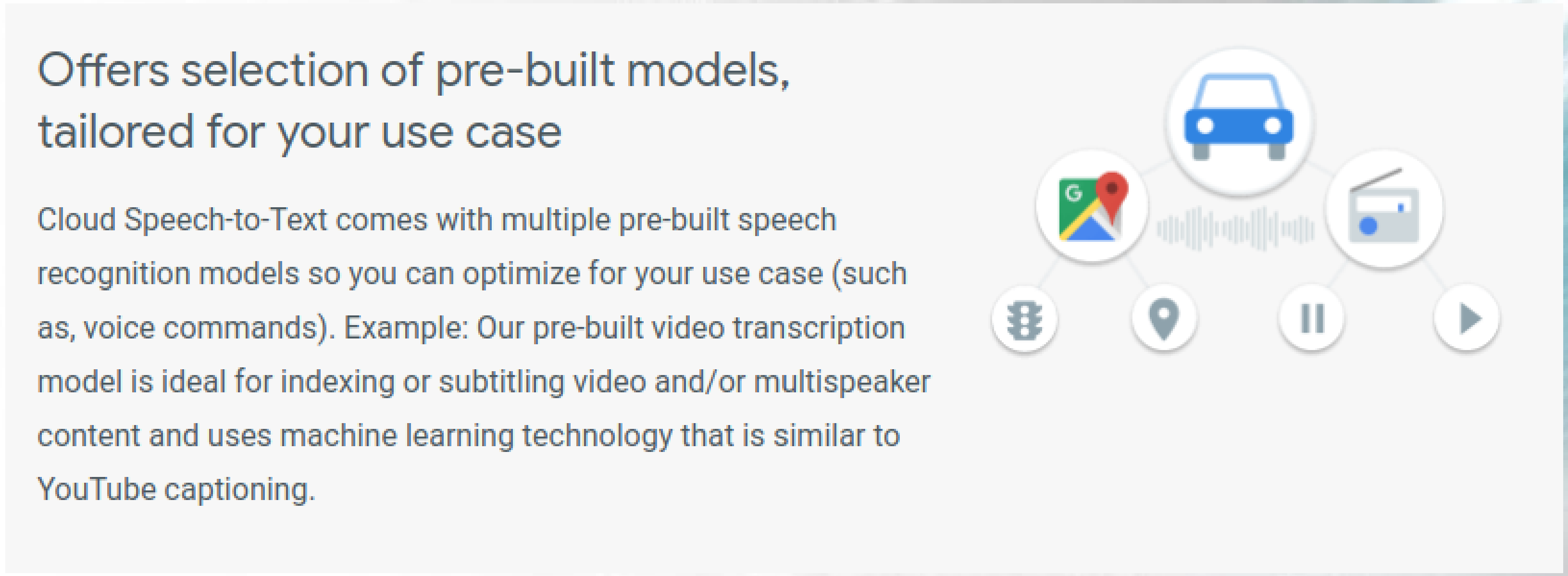
Input: Digital audio of user's voice

Output: Text version of user's spoken words

GENIVI®

VCIVING Pipeline Step 2: Speech-To-Text

This project used Google Cloud Speech-To-Text API together with a third-party library, but also looked at technology from Mozilla. The goal was to be agnostic to where the pre-trained model resided



Offers selection of pre-built models, tailored for your use case

Cloud Speech-to-Text comes with multiple pre-built speech recognition models so you can optimize for your use case (such as, voice commands). Example: Our pre-built video transcription model is ideal for indexing or subtitling video and/or multispeaker content and uses machine learning technology that is similar to YouTube captioning.

GENIVI®

VCIVING Pipeline Step 3: Interpretation

- The textual representation of the audio data received from step 2, is interpreted.
- The underlying meaning (the gist) of the user’s speech is refined and converted to a Task/Process/Function in the IVI system
- A pre-trained model which contains a wide-variety of different phrases through which the user can execute the IVI commands, should be used.
- Neural network model training requires lots of data to be effective as well as ample computing power.

GENIVI®

VCIVING Pipeline Step 3: Interpretation

- The model should accept the phrases/sentences as the input and map them to a TPFIVIS.
- Instead of directly addressing TPFIVIS, a wrapper should be used where it handles the interactions with IVI and also directly providing outputs based on the responses from IVI.

Input

- The textual representation of user’s speech.

Output

- Mapping to a TPFIVIS wrapped function.

GENIVI®

VCIVING Pipeline Step 4: Execution

- After the user's speech has successfully been interpreted, the wrapper function should be called subsequently.
- The wrapper function should,
  - Execute the TPFIVIS(by interacting with IVI systems directly).
  - Provide outputs to the user regarding the status of execution.
  - Handle every necessity after it is called.
    - For example: If more information is required, implementing features like continued conversation with user.

VCIVING Pipeline Step 4: Execution

- Since wrapper function is handling the execution after it gets called, there is no output from this step.
- The wrapper functions will handle the output by itself.

Input

- Mapping to a TPFIVIS wrapped function.

Output

- (No Output)

EmulationCore

- EmulationCore is the implementation of VCIVING on COTS.
- It's name derives from the idea that this emulates a complete system on target
- It performs the same tasks;
  1. Captures inputs from the user.
  2. Recognizes and interprets speech to text
  3. Executes the task which is meant by the input.

References

- **Project Wiki and Documentation**:
  - https://bit.ly/2R8X6c0
- **Project Repository:**
  - https://github.com/GENIVI/GENIVI-GSoC-18
  - Build Instructions: https://bit.ly/2Io73yk
- **Initial Project Idea**
  - https://bit.ly/2N915Cb
  - Section: Voice command of IVI system
- **Comprehensive Explanations, Guides and Documentation**
  - EmulationCore: https://bit.ly/2R8X6c0
  - Input Handling and Input Handler: https://bit.ly/2IpDrAt
  - Output Handling and Output Handler: https://bit.ly/2Iror5h
  - **Task Executors**: https://bit.ly/2y1J3w4

Functions of EmulationCore

1. The main focus of the EmulationCore would be to capture and control the inputs and their flow and to handle the outputs.
   - Example: Once user speaks, the input is grabbed through the microphone by listening consistently, performs Speech Recognition to convert audio data to its textual representation, finally interpreting the data.

Functions of EmulationCore

2. Upon interpretation, the execution of the underlying TPFIVIS is handled over to the set of wrapper classes: TaskExecutors.
    - Example: The final interpretation from EmulationCore will be a mapping to a method in one of the wrapping classes of TaskExecutors. Subsequently, the respective method will be executed.

VCIVING Pipeline in EmulationCore

1. **Step 1: Grabbing Inputs**
   - Input Mechanisms are used to listen and read inputs from Input Devices.
2. **Step 2: Speech-To-Text**
   - Third party APIs and libraries are used to convert the audio data to its textual representation(Speech Recognition).

VCIVING Pipeline in EmulationCore

3. Step 3: Interpretation

A pre-trained model is utilized to understand the user speech.

4. Step 4: Execution

Wrapper classes, wrapped around TPFIVIS are used.

VCIVING Pipeline on EmulationCore: Step 1 – Grabbing Inputs

- The primary concern of this step is to collect/grab information from the input devices.
- The Input Devices are wrapped inside another set of wrapper classes: InputMechanisms.

VCIVING Pipeline on EmulationCore: Step 1 – Grabbing Inputs

Each an every Input Device is wrapped inside a separate InputMechanism class. Depending on how each InputMechanism reads input data from the Input Device, implementation is slightly different with several characteristic methods.

Example for Step 1: Microphone

- Microphone is the major Input Mechanism in the EmulationCore.
- When the microphone is initialized, functions are defined to,
  - Wait for input from the microphone.
  - Transfer the captured audio data to Step 2(Speech-To-Text).

Example for Step 1: Microphone

Since interpretation of the speech is done under another protocol(Grabbers and GrabberController), it is not implemented in the wrapper class.

Listening to the microphone and processing is carried out on a separate thread.

VCIVING Pipeline on EmulationCore: Step 2 – Speech-To-Text

- Audio data received from the microphone is converted to textual representation.
- Google Cloud Speech-To-Text API was used together with a third-party library in order to facilitate the process.

VCIVING Pipeline on EmulationCore: Step 2 – Speech-To-Text

Audio data is sent to Google STT API which would return back the textual representation. If useful speech is not found in audio data, an error will be thrown by Google APIs which is handled by notifying the user.

VCIVING Pipeline on EmulationCore: Step 3 – Interpretation

- A model is trained to interpret/refine the underlying command related to a TPFIVIS.
  - Training data: Training dataset would contain a phrase by which the user implies a certain command, as the feature and the mapping to a method in a wrapping class of TaskExecutors, as the label, per each example.

VCIVING Pipeline on EmulationCore: Step 3 – Interpretation

A bag of words is maintained by the algorithm for encoding into numeric data.
It is used to convert text data to numerical format.
Finally the model is trained to accept textual input and return a mapping to existing TaskExecutor method.

VCIVING Pipeline on EmulationCore: Step 3 – Interpretation

- The textual data is passed through the described model.
- This would provide us a TaskExecutor name followed by a method inside it.
- This combination is known as a Mapping.
- Mapping is returned through this step.

VCIVING Pipeline on EmulationCore: Step 4 – Execution

- The respective mapping to the method(from Step 3) is converted to a callable function(in python).
- This function would collect data by calling other different functions and methods which are required to execute TPFIVIS.
- Ultimately, the method will be executed,
  - In a separate thread
  - Passing required data as arguments.

Workflow of EmulationCore

EmulationCore proceeds through the following steps.

a. Reading the Settings file

Settings file is a JSON file where all required parameters required for EmulationCore are stored.

A SettingsContainer is used to pass the settings throughout the program.

Workflow of EmulationCore

b. Initialization of I/O Handlers

Input Handler defines and controls all the Input Mechanisms(Wrappers around Input Devices such as Microphone).
TaskExecutors(Wrappers around TPFIVIS) are loaded dynamically into the program during the initialization of Input Handler.
Output Handler defines and controls all the Output Mechanisms(Wrappers around Output Devices such as Speaker) while handling a queue-based mechanism.

Workflow of EmulationCore

c. Initialization of Text-Input Handler

Text-Input handler is merely for debugging purposes.
Inputs through the console are accepted and currently supports several commands to bypass microphone inputs, output data from speaker etc.

Comprehensive Explanation:

https://bit.ly/2R8X6c0

GENIVI®