An LLM-Agent-based RTL Code Generation Augmentation System

Senior Design 1 Preliminary Design Proposal

Prepared by Group G-05:

Jacobo Forero

Caleb Elliott

Andrew Chambers

Sammy Fares

Mateus Verffel Mayer

Dexter Pressley

Prepared for:

Senior Design 1

Department of Computer Science

University of Central Florida

October 31, 2025

Table of Contents

Chapter 1: Introduction

1.1 The Challenge of Modern Hardware Design

For over half a century, the semiconductor industry has been driven by the relentless pace of Moore's Law, leading to an exponential increase in the number of transistors that can be integrated onto a single chip. While this has enabled unprecedented computational power, it has also given rise to a critical challenge: the "design gap." System complexity has begun to outpace the productivity of human engineers, creating a significant bottleneck in the hardware design lifecycle. The process of translating a high-level functional specification into a correct, efficient, and verifiable hardware implementation remains a profoundly manual, time-intensive, and error-prone endeavor.

The heart of this challenge lies at the Register-Transfer Level (RTL), the primary abstraction at which most digital hardware is designed. While higher-level synthesis (HLS) tools exist to generate RTL from languages like C++ or SystemC, a vast amount of critical design work is still performed

directly in Hardware Description Languages (HDLs) such as SystemVerilog. As noted by researchers at the 2023 ACM/IEEE Design Automation Conference (DAC), this direct engagement with RTL is essential for achieving optimal performance, power, and area (PPA), but it represents a major engineering bottleneck where designers must manually manage complex state machines, intricate data paths, and critical timing constraints. This manual process initiates a tedious and iterative cycle: write RTL code, develop a corresponding testbench, run simulations, meticulously analyze waveforms to debug failures, and refine the code. A loop that consumes the majority of a project's timeline and budget.

The recent advent of powerful Large Language Models (LLMs) presents a transformative opportunity to address this long-standing productivity gap. By leveraging their advanced reasoning and code generation capabilities, it is now possible to envision an automated system that can act as a highly capable assistant to hardware engineers, automating the most laborious aspects of RTL generation and verification. This project is an exploration and implementation of such a system.

1.2 Project Vision and Core Principles

To address the challenges of RTL design, our project introduces a multi-agent system designed to accelerate and automate the hardware design lifecycle. The vision is not to replace the human engineer, but to create a powerful, transparent, and reliable tool that allows them to focus on high-level architectural decisions while delegating the mechanical aspects of implementation and verification to a team of specialized AI agents.

Our entire system architecture is founded on a single, guiding principle: **Exhaustive upfront planning enables mechanical, parallel execution.**

We posit that the ambiguity and "hallucinations" often associated with AI-driven generation can be drastically mitigated by refusing to delegate open-ended creative decisions to the agents. Instead, we invest heavily in a human-supervised planning phase where all architectural ambiguities, interface contracts, and verification goals are resolved *before* any implementation begins.

This core principle manifests in two key architectural decisions that define our approach:

1. **A Strict Two-Phase Workflow:** The system is bifurcated into a strategic **Planning Phase** and a tactical **Execution Phase**.
    - In the **Planning Phase**, a human designer collaborates with a Specification Helper Agent to produce a complete and unambiguous design specification. This frozen plan, represented as a Directed Acyclic Graph (DAG), becomes the immutable source of truth.
    - In the **Execution Phase**, an Orchestrator dispatches well-defined, unambiguous tasks from the DAG to a pool of agents. Because the plan is complete, the agents' work becomes largely mechanical and highly parallelizable, transforming them from uncertain explorers into skilled executors.
2. **The Agent vs. Process Duality:** We recognize that not all tasks require the stochastic creativity of an LLM. Our system therefore utilizes two fundamental primitives:
    - **Agents** are employed for tasks requiring reasoning, generation, and creative problem-solving, such as writing RTL code (Implementation Agent) or hypothesizing the root cause of a bug (Debug Agent).

- **Processes** are used for deterministic, verifiable tasks, such as linting code, compiling a design, or running a simulation.

By adhering to these principles, our architecture transforms the hardware design process from an unpredictable art into a structured, manageable, and highly automated workflow. This approach is designed to build trust, ensure predictability, and ultimately accelerate the journey from concept to verified silicon.

1.3 Minimum Viable Product and Definition of Done

To ground our ambitious vision in a tangible set of deliverables, the team established a clear Minimum Viable Product (MVP). The MVP defines the essential features required to create a functional and valuable tool by the end of the project, while also outlining future enhancements as stretch goals. This scope provides a clear "definition of done" for our development efforts and ensures we remain focused on delivering a core, high-impact system.

The following features comprise the MVP and will be implemented in the initial version of the tool:

1. **Command-Line Interface (CLI) Tool**
    - A user-friendly CLI that allows engineers to interact with the system, define prompts, and configure generation parameters.
    - Supports both direct user and scriptable interaction, allowing for flexible usage.
2. **LLM Agent Integration**
    - Uses API access to one or more LLMs for code generation.

- The system leverages multiple specialized “agents” that handle different stages of code generation (e.g., verification, optimization, code review).
- Incorporate a maximum iteration count setting to cap the number of prompt-response cycles before pausing code generation and validation.

3. **Functional Verilog Code Output**
    - Generates syntactically correct and logically functional Verilog code.
4. **Built-in Evaluation and Testing**
    - Automatic agent-based code validation that generates and runs testbenches to verify generated Verilog code.
    - Ensures output meets functional correctness and minimal performance standards.
5. **Chain-of-Thought Traceability**
    - Every output includes a detailed breakdown of the LLM’s reasoning process from the initial prompt to the final code.
    - This helps engineers understand intermediate steps, assumptions made by the model, and the rationale behind architectural decisions.

Upon the successful implementation of the core MVP, the team has identified several high-impact features that would significantly enhance the system's capabilities. These are defined as stretch goals:

1. **Configurable Iteration Control**
    - Engineers can review intermediate results and guide the next steps if needed.
2. **Interactive Pause and Query System**

- Enables LLM agents to pause and request clarification or guidance from the engineer if they encounter ambiguity or logical roadblocks.
- Adds a semi-supervised layer to improve code quality and reduce hallucinations.

3. **Graphical User Interface (GUI)**
    - A visual frontend for interacting with the system, including prompt configurations, code inspection, and visualization of chain-of-thought outputs.
    - Intended to improve accessibility for users less comfortable with CLI tools.
    - Supports richer debugging and workflow management.
4. **Synthesis**
    - Synthesis checks and self-correction in addition to functional correctness.

1.4 A "Golden Path" Example: The 4-Bit Counter

To make the abstract principles of our architecture concrete, this paper will use a simple, recurring example: a "golden path" to illustrate how a design progresses through the system. This example is a standard digital logic component: **a 4-bit synchronous up-counter with an active-high asynchronous reset and a count-enable input.**

A human engineer would initiate the process with a simple natural language prompt, such as: *"Create a 4-bit synchronous counter. It should increment on the positive clock edge only when the 'enable' signal is high. It needs an active-high asynchronous reset that sets the count to zero."*

This seemingly simple request is immediately processed by our two-phase workflow:

1. **Phase 1 (Planning):** The prompt is first handled by the Specification Helper Agent. Instead of immediately generating code, the agent begins a clarification dialogue to resolve ambiguities inherent in the request, demonstrating our "exhaustive upfront planning" principle. It might ask:
    - *"What should the counter's behavior be when it reaches its maximum value of 1111 and is enabled? Should it wrap around to 0000 or saturate?"*
    - *"What is the desired name for the clock, reset, enable, and 4-bit output ports?"*
2. Through this dialogue, a complete and unambiguous specification is formed. The Planner Agent then consumes this frozen specification to generate a simple Directed Acyclic Graph (DAG). For this design, the DAG would consist of a single module node for the counter and a corresponding testbench node, with a dependency showing that the testbench requires the counter's interface to be frozen.
3. **Phase 2 (Execution):** Once the plan is approved, the Orchestrator begins execution. It identifies that the counter module node is in the Stub state and its dependencies are met. It then creates and publishes a TaskMessage to the agent_tasks queue, instructing an Implementation Agent to generate the RTL code. Upon successful generation, the module's state transitions to Draft, which in turn makes it eligible for a test task.

Throughout the subsequent chapters, we will revisit this 4-bit counter example to provide a practical illustration of more complex concepts,

including the structure of a TaskMessage, the flow of work through the Task Broker, the use of agent memory during debugging, and the final evaluation of the generated artifacts.

Chapter 2: Background and System Philosophy

2.1 State of the Art in AI-Aided Hardware Design

The field of AI-driven hardware design has seen rapid advancements, moving from simple code generation to sophisticated, multi-agent frameworks. Before designing our own architecture, our team conducted a thorough review of contemporary and state-of-the-art approaches to understand the prevailing challenges and successful strategies. This research provided the foundational knowledge upon which our system is built.

Several key frameworks highlight the industry's trajectory. The Spec2RTL-Agent from NVIDIA represents a significant step towards fully automated spec-to-code generation. Its architecture introduces a reasoning module for parsing specifications, a progressive coding module for iterative refinement, and a "reflection" module for self-debugging. Notably, it generates an intermediate C++ representation for High-Level Synthesis (HLS), a two-stage approach that ensures greater correctness and synthesizability than direct-to-Verilog methods. This system demonstrated a remarkable reduction in the need for human intervention by up to 75%, validating the power of multi-agent planning and intermediate representations.

Similarly, the "Agentic AI" framework from Infineon (Gadde et al.) emphasizes a structured, multi-agent workflow for end-to-end design and verification. This approach utilizes parallel agent streams to generate both RTL code and formal properties (SystemVerilog Assertions) from a unified plan. A key innovation is its deep integration with commercial EDA tools for linting and formal verification within the agent loop, allowing "critic" agents to provide targeted feedback beyond simple simulation results. This comprehensive, deliberative process achieved over 95% functional coverage on open-source benchmarks, proving that a collaborative agent system with integrated formal methods is both adaptable and highly effective.

As a foundational baseline, the AIVRIL framework was one of the earlier systems our team analyzed. It effectively demonstrated the core value of keeping verification "in the loop." By having one agent generate code while another continuously simulates and validates it, AIVRIL could catch and fix errors that a single-pass LLM would miss, achieving an 88.5% success rate on the VerilogEval-Human benchmark. While later frameworks have expanded on this concept by incorporating more advanced verification techniques, AIVRIL provided a strong proof-of-concept for iterative, feedback-driven refinement.

Finally, the VFlow framework introduced a novel and powerful concept: optimizing the agent workflow itself. Instead of hand-crafting the sequence of agent and tool interactions, VFlow treats the entire process as a search problem, using a variant of Monte Carlo Tree Search to discover the optimal sequence of prompting, simulation, and refinement steps. This approach led to a 20-30% higher success rate and demonstrated that even smaller LLMs could outperform larger ones when orchestrated intelligently, achieving up to 10.9x better cost-efficiency.

2.2 From Research to a Design Philosophy

Our analysis of these state-of-the-art systems revealed several common themes that directly shaped our design philosophy. We observed that the most successful frameworks converged on a set of core ideas, which we adopted and, in some cases, extended.

First, it was clear that one-shot generation is insufficient. Every successful modern framework, from AIVRIL's simple feedback loop to Infineon's complex deliberation, embraces iterative refinement. This insight is the foundation of our automated Execution Phase, where artifacts are progressively improved through a cycle of generation, testing, and debugging.

Second, the use of a multi-agent system (MAS) is a validated strategy. The specialized roles in the NVIDIA and Infineon frameworks demonstrated that dividing responsibilities such as planning, implementation, and verification leads to better outcomes than relying on a single monolithic model. This directly informed our decision to create a team of distinct agents, each with a clearly defined purpose, such as the Implementation Agent, Testbench Agent, and Debug Agent.

Third, the importance of structured, hierarchical planning was a recurring motif. The success of hierarchical prompting and the planning agents in the Spec2RTL and Infineon systems motivated our most critical architectural decision: to create a strict separation between planning and execution. We concluded that the most effective way to manage the complexity and non-determinism of LLMs was to resolve as much ambiguity as possible before any code is written. This led to our core principle of "exhaustive upfront planning," where we insist on a fully specified and frozen Directed

Acyclic Graph (DAG) as the immutable blueprint for the automated execution phase.

Finally, the research highlighted the necessity of deep integration with traditional EDA tools. The Infineon framework's use of formal verification and linting proved that relying solely on simulation is a limited approach. This validated our "Agent vs. Process Duality," where we explicitly distinguish between creative agent tasks and deterministic process tasks, and led to the design of a generic Tool Executor to manage these essential, non-agentic verification steps.

2.3 The Human-in-the-Loop Imperative

While the ultimate goal of these systems is automation, our research reinforced the belief that the human engineer is an indispensable part of the process. Many frameworks, like Infineon's, utilize a human-in-the-loop model for resolving impasses when agents get stuck. However, we believe this reactive approach, while necessary, is insufficient for building a truly reliable and trustworthy tool.

Our architecture elevates the human role from a mere troubleshooter to a strategic partner in the upfront planning phase. The core of our Human-in-the-Loop model is proactive collaboration, not reactive intervention. This is embodied by the Specification Helper Agent, whose primary function is to engage in a dialogue with the human designer to converge on a complete and unambiguous specification.

The rationale for this decision is twofold. First, by forcing all ambiguities to be resolved before the automated phase begins, we drastically reduce the likelihood that agents will get stuck due to misinterpretation or incomplete

requirements. This minimizes the number of costly debugging cycles and prevents the system from making unguided, potentially incorrect architectural assumptions. Second, this collaborative planning process builds essential trust. The engineer remains in full control of the design's strategic direction, using the system as a powerful tool to execute their precise intent, rather than as a "black box" that produces an inscrutable result. By positioning the human as the ultimate authority on the plan, we ensure the final output is not just functionally correct, but is also the product the engineer intended to build.

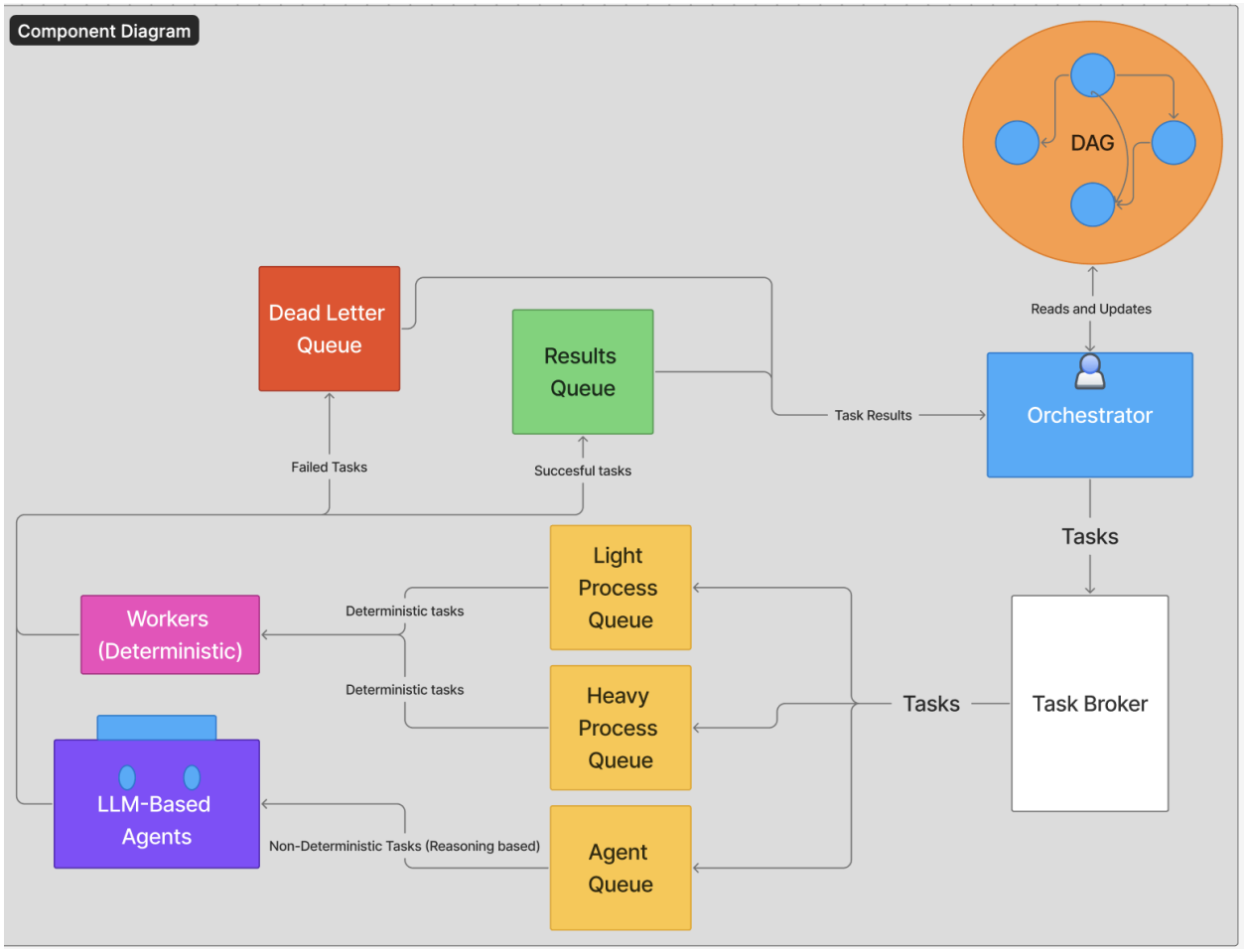
Chapter 3: The Two-Phase System Architecture

The architecture of our multi-agent hardware design system is the direct embodiment of our core principle: exhaustive upfront planning enables mechanical, parallel execution. This chapter provides a detailed examination of this architecture, beginning with a high-level overview of its primary components and then delving into the distinct workflows, state transitions, and agent roles that define its operation.

3.1 High-Level Overview

At its core, the system is a distributed, asynchronous platform composed of four primary components that work in concert to manage the hardware design lifecycle. These components are the Orchestrator, the Task Broker, the Worker Pools, and the Data Stores. Their interaction is designed to decouple strategic decision-making from tactical execution, allowing for maximum parallelism and resilience.

Figure 3.1: System Architecture Component Diagram



- **The Orchestrator** acts as the central "brain" of the system during the execution phase. It does not perform any design tasks itself; its sole responsibility is to maintain the state of the entire design graph (DAG),

identify which tasks are ready to be executed based on their dependencies, and publish those tasks to the Task Broker.

- **The Task Broker** is the communication backbone of the system, implemented as a message bus. It decouples the Orchestrator from the workers, managing a set of queues that buffer tasks for different types of work. This ensures that long-running tasks do not block short ones and allows the system to scale by simply adding more workers.
- **The Worker Pools** are collections of specialized processes that consume tasks from the Task Broker and perform the actual work. These are divided into **Agent Pools** for creative, LLM-driven tasks (like writing code) and **Process Pools** for deterministic tasks (like running a compiler or simulator).
- **The Data Stores** serve as the system's memory. The **Design Context** is a read-only store containing the frozen output of the planning phase, while the **Task Memory** is a writeable store that records the attempts, logs, and reflections for each task, enabling stateful, iterative debugging.

3.2 The Two-Phase Workflow

The system's operation is strictly divided into two distinct phases. This separation is a hard gate that ensures all strategic, architectural, and ambiguous decisions are resolved with human oversight before any automated, parallel execution begins.

This initial phase is a collaborative, human-supervised process focused on transforming a high-level design intent into a complete, unambiguous, and machine-executable plan. It involves two key agents:

1. **Specification Convergence:** The process begins with a human designer interacting with the Specification Helper Agent. Through a guided, conversational workflow, the agent ensures that all aspects of the design from functional intent and interface contracts to the verification plan are fully defined and free of ambiguity.
2. **DAG Construction:** Once the specification is complete and "frozen," the Planner Agent takes over. It performs an automated, recursive decomposition of the design, breaking it down into a Directed Acyclic Graph (DAG) of modules, interfaces, and testbenches. This process continues until every leaf node in the graph represents a simple, implementable unit or a component from a standard library.

The final output of this phase is the **Frozen Design Context**, an immutable "source of truth" containing the complete DAG and all associated specifications. This context must be formally approved by the human designer before the system can proceed to the next phase.

This phase is fully automated and driven by the orchestration loop. It is a continuous cycle of discovering ready tasks and dispatching them for parallel execution:

1. **Task Discovery:** The Orchestrator constantly scans the DAG in the Design Context to find nodes whose dependencies have been met, making them ready to transition to a new state. For example, a module in the Stub state is ready for an implement task.
2. **Task Publication:** For each ready node, the Orchestrator constructs a TaskMessage and publishes it to the appropriate queue on the Task Broker.

3. **Task Consumption & Execution:** An available worker from the corresponding pool consumes the task message and begins execution, entirely independent of the Orchestrator and other workers.
4. **Result Notification & State Update:** Upon completion, the worker publishes a ResultMessage. The Orchestrator consumes this result, updates the state of the corresponding artifact in the DAG, and the loop repeats. A successful implementation, for instance, transitions a module to the Draft state, which may immediately make it eligible for a test task in the Orchestrator's next scan.

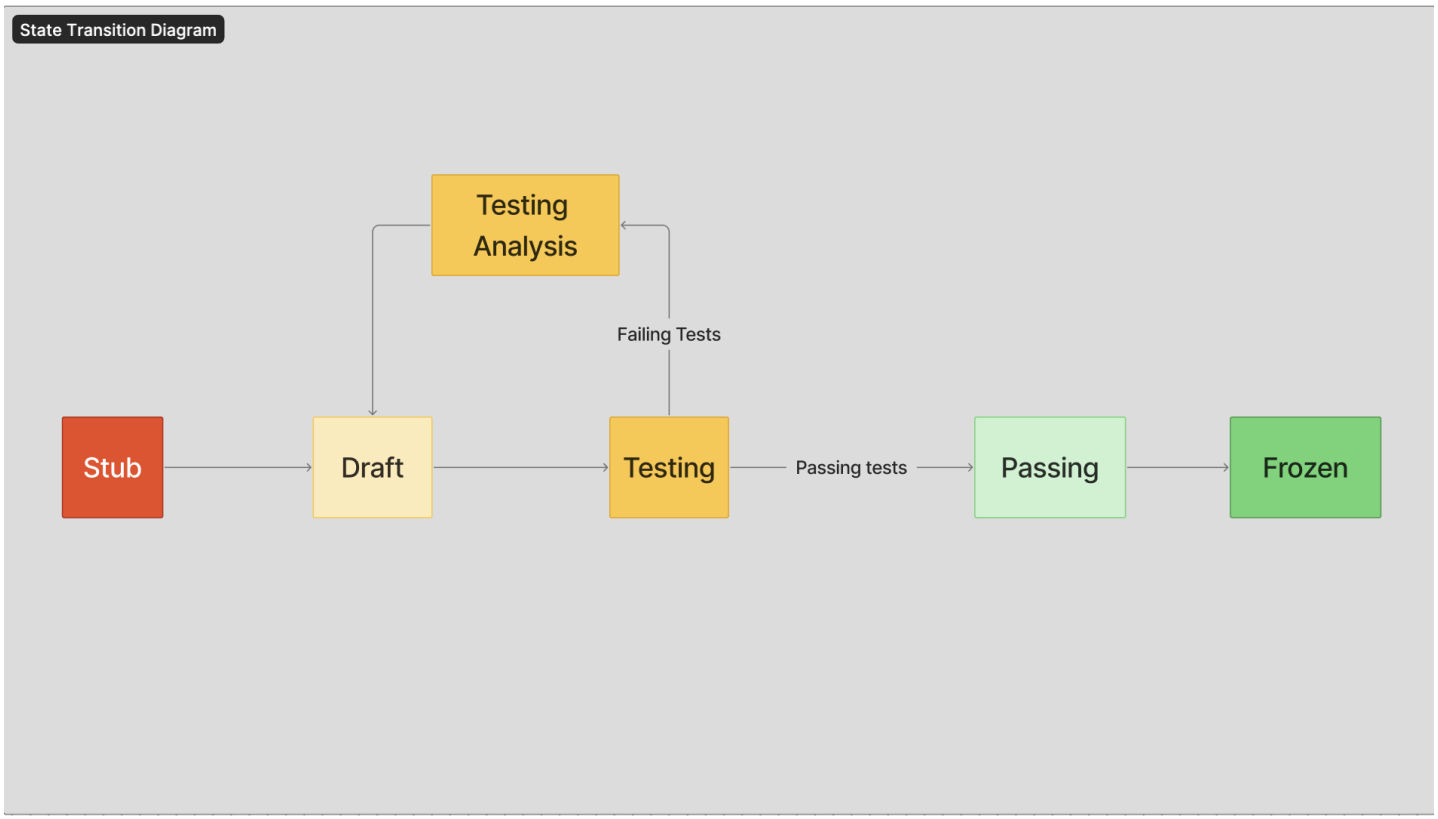
3.3 The Artifact State Machine

To manage progress in a structured and predictable way, every design artifact within the system (e.g., an RTL module, a testbench) exists within a formal state machine. A task, therefore, is an action intended to transition an artifact from one state to the next. The primary states in an artifact's lifecycle are:

- **Stub:** The artifact is defined in the plan. Its specification and interfaces exist, but no implementation has been generated.
- **Draft:** An initial version of the implementation has been generated by an agent, but it has not yet been subjected to verification.
- **Testing:** The artifact is actively undergoing verification (e.g., a simulation is running against it).
- **Testing_Analysis:** A test has failed, and the artifact has entered a multi-stage pipeline for detailed failure analysis and debugging.
- **Passing:** The artifact has successfully passed all its required verification checks.

- **Frozen:** The artifact is in a Passing state, and its interfaces are locked. It can now be safely used as a dependency by other components in the design.

Figure 3.3: State Transition Diagram



3.4 The Testing_Analysis Pipeline

When a verification task fails, the system does not simply loop back to a generic debug step. Instead, the artifact enters the specialized Testing_Analysis state to undergo a sophisticated, three-stage ordered pipeline designed to provide the Debug Agent with the richest possible context for performing a targeted fix.

1. **Stage 1: Distillation:** The process begins with a distill task, executed by a deterministic worker from the Process Pool. This task takes the large, raw outputs of a failed simulation (such as waveform files and verbose logs) and reduces them to a small, failure-focused dataset. This crucial step extracts only the relevant signals and time windows related to the failure, dramatically reducing the amount of data the subsequent AI agent needs to analyze.
2. **Stage 2: Reflection:** Once the data is distilled, a reflect task is dispatched to the Reflection Agent. This LLM-based agent analyzes the distilled dataset to generate structured insights. Its output is not corrected code, but rather a set of hypotheses about the root cause of the failure, a list of likely failure points in the source code, and recommended investigation paths.
3. **Stage 3: Enhanced Debugging:** Finally, a debug task is sent to the Debug Agent. This agent receives the complete failure context: the original failing code, the distilled dataset from Stage 1, and the structured insights from Stage 2. Armed with this rich, pre-processed information, the agent can perform a much more targeted and effective code correction.

If the debug attempt is successful and subsequent tests pass, the artifact moves to the Passing state. If not, it may re-enter the analysis pipeline or, after a set number of attempts, be escalated to a human expert.

3.5 Agent Definitions and Roles

The system employs a team of six specialized agents, each designed with specific responsibilities and operating within a designated phase of the workflow. The clear division of labor ensures that each agent can be optimized for its particular task, from high-level strategic planning to targeted, tactical bug fixing. A detailed description of each agent's architecture and responsibilities can be found in the docs/Agents.md document. The following table provides a high-level summary of their roles.

Agent	Phase	Primary Input	Primary Output	Core Function
Specification Helper Agent	Planning	Human Drafts	Frozen L1–L5 Specification	Specification Convergence
Planner Agent	Planning	Frozen L1–L5 Specification	Design Context (DAG)	Strategic Decomposition
Implementation Agent	Execution	Module Specification & Interface	RTL Code (.sv)	Code Generation
Testbench Agent	Execution	Verification Plan	Testbench	Verification Generation

		& Interface	Code (.sv)	
Reflection Agent	Execution	Distilled Data & Failure Context	Debugging Insights	AI-Powered Analysis
Debug Agent	Execution	Failing Code & Analysis Artifacts	Corrected RTL Code	Targeted Bug Fixing

Chapter 4: Agent Memory

4.1 The Need for an Evolving Memory System

Traditional memory systems for large language model (LLM) agents have clear limits. Most function as passive storage, keeping information exactly as it was first stored and retrieving it using fixed rules. This approach is sufficient for short or simple interactions, but it breaks down when tasks are long, multi-step, or technically complex. In a real engineering workflow, such as debugging or refining Verilog, an agent must remember not only the most recent step but also prior mistakes, previous fixes, and specific design patterns. Basic memory systems cannot support this because once information is stored, it never changes, improves, or interacts with new knowledge. For our project, this is a serious limitation, as the RTL design loop is not a one-shot task; it is iterative, cumulative, and requires context that evolves over time.

To address this problem, we have based our memory design on the principles of "agentic memory" systems like A-Mem, which treat memory as an active and improvable component. Instead of storing information like a

static database, this approach allows memory to evolve based on new experience. It is inspired by the Zettelkasten note-taking method, where each idea is saved separately and linked to related ideas over time. In our system, each new agent interaction is turned into a structured note that includes the original content, timestamps, keywords, and a short LLM-generated explanation of context. These notes are not isolated; the system automatically links them to similar past memories, forming a connected network of knowledge. This is critical for our project because RTL refinement benefits heavily from pattern linking. If an agent has seen a similar testbench failure or a previous state machine error, that memory must be retrievable in a way that reflects its relation to the new problem, not just its raw text.

Agentic memory also updates past memories when new information appears. If an agent learns a better solution to a recurring bug, the system connects that new solution to old, related notes and can even revise older entries so that archived knowledge reflects the updated understanding. This is closer to human learning, where new insight reshapes how we interpret the past. For our use case, this is essential: once a better RTL fix or coding standard is established, the agent must not retrieve outdated logic from older sessions.

4.2 The Three-Tier Memory Architecture

Our system's architecture defines a clear separation of memory stores that aligns with the need for both stable, long-term context and dynamic, short-term learning. We conceptualize this as a three-tier architecture: Design Context (Long-Term), Task Memory (Short-Term), and Working Memory (In-Context).

As defined in our system architecture, the **Design Context** serves as the system's long-term, immutable memory. It contains the complete, frozen output of the planning phase, including the final specification, the dependency graph (DAG), and all module interfaces. This read-only data store acts as the "source of truth" for the entire execution phase, ensuring that all agents operate from a consistent and stable foundation. It prevents preference drift and self-contradiction across sessions by providing a canonical reference for all strategic decisions.

The **Task Memory** is the writeable, structured store that captures the dynamic state of the execution phase. This corresponds to the system's short-term memory, holding artifacts for each task, such as code attempts, simulation logs, distilled datasets, and reflection outputs. This is where the principles of agentic memory are applied. We selected this design to make the system responsive and focused. It prevents agents from being overwhelmed by their entire history while still allowing them to transfer key insights to long-term memory once a fix is confirmed.

We further enhance this with concepts from Mem0, a system designed for fast, lightweight memory for current reasoning. Mem0 works in two phases: extraction and update. During extraction, the agent identifies new facts from the latest exchange (e.g., a failed test case). During the update, it compares these facts to existing items in Task Memory and decides whether to merge, replace, or discard them. This continuous maintenance prevents memory clutter and keeps the agent’s understanding current.

Working memory is the most transient layer, representing the context window provided to an LLM agent for a single, specific task. When the Orchestrator dispatches a task for instance, to the Debug Agent. It populates the agent's context with highly relevant, filtered information retrieved from

the Design Context and Task Memory. This "mental scratchpad" is analogous to human short-term memory, allowing the agent to focus exclusively on the immediate problem. Once the task is complete, the key results are written back to Task Memory, and the working memory is discarded.

4.3 Memory in Action: Supporting the Reflexion Pattern

Our architecture specifies a formal Testing_Analysis state that implements a learning loop inspired by the Reflexion pattern. This workflow, involving distill → reflect → debug tasks, relies heavily on the Task Memory design.

The process unfolds as follows:

1. **Distillation:** A deterministic process analyzes raw failure logs (e.g., waveforms) and extracts a concise, failure-focused dataset. This summary is written to the **Task Memory**.
2. **Reflection:** The **Reflection Agent** receives this distilled data. It analyzes the summary to produce debugging hints and root-cause hypotheses. These structured insights are also written back to the **Task Memory**, linked to the original failure.
3. **Debugging:** The **Debug Agent** is then invoked. It retrieves the failing code, the distilled logs, and the reflection insights from **Task Memory**. This context allows the agent to propose a targeted code fix. Crucially, the agent also reviews its own past attempts stored in Task Memory to avoid repeating mistakes.

This cycle where attempts and analysis are logged, reflected upon, and used to inform the next action is the core learning mechanism of our system. The Task Memory is specifically designed to store these attempts and logs, enabling the Debug Agent's iterative learning loop.

4.4 Retrieval-Augmented Generation (RAG): Bridging Memory and Action

Having a well-structured memory is not enough; the system must also find the right information efficiently. Retrieval-Augmented Generation (RAG) acts as the bridge between what the system knows (stored in the Design Context and Task Memory) and what an agent needs at a given moment.

RAG combines search with generation by first finding relevant documents and then feeding them into the LLM to guide its response. We employ advanced RAG techniques to ensure accuracy in our technical domain:

- **Hybrid Search:** We use a mix of semantic search (for conceptual matches like “signal not driven” vs. “floating net”) and keyword search (for exact signal or module names). This is vital in hardware design, where problems involve both concepts and precise syntax.
- **Query Rewriting:** Before searching, the system can refine a vague query like “why is my FSM stuck?” into concrete sub-questions, such as “are all state encodings reachable?” This decomposition improves retrieval accuracy.
- **Response Synthesis:** Instead of dumping raw text into the agent's context, RAG filters, summarizes, and compresses it. This prevents distraction, reduces latency, and increases faithfulness to the source evidence.

In our system, RAG is the intelligent librarian for our memory stores. It ensures that when an agent needs information, it receives only the most relevant, context-matched sources.

4.5 Framework Evaluation and Design Choices

To implement RAG efficiently, we evaluated four major frameworks: LlamaIndex, LangChain, Haystack, and Pydantic.

Framework	Primary Role	Strengths	Weaknesses	Relevance to Project
LlamaIndex	Data indexing & retrieval for RAG	Fast, lightweight, flexible with mixed data types	Smaller community than LangChain	Chosen core for RAG: best balance of retrieval accuracy + simplicity for Verilog data.
LangChain	LLM workflow orchestration	Rich ecosystem, strong tool integration	Higher complexity, slower for pure retrieval	Considered for future orchestration but not for current retrieval layer.
Haystack	Search & QA pipeline	Enterprise-grade, supports hybrid retrieval	Heavy setup, resource-intensive	Useful for later production deployment, not prototype scale.
Pydantic	Data validation & schema definition	Enforces clean structured data, easy integration	Minor performance overhead	Used to validate memory and Task Memory entries.

We chose **LlamaIndex** as the retrieval foundation because its clean, high-performance API integrates smoothly with our memory architecture. Its modular design allows the system to embed Verilog modules, documentation, and logs into a searchable index and query them efficiently. **Pydantic** complements this by enforcing structured, validated data exchange between agents, ensuring the data that flows between memory

and the agents remains clean and consistent, which is critical for preventing "poison pill" tasks.

This combination of an evolving agentic memory (**Task Memory**), an intelligent retrieval mechanism (**RAG with LlamaIndex**), and a stable, shared workspace (the **asynchronous architecture**) was chosen to mirror how expert engineering teams operate. The system remembers what matters, collaborates transparently, and continuously improves.

4.6 System-Level Benefits of the Architecture

This memory and retrieval design was chosen not just for technical novelty but because it directly supports the primary goals of our multi-agent system: scalability, reliability, and efficiency.

- **Scalability:** The architecture supports growth. Because agents interact asynchronously through the Task Broker and shared Data Stores, new agents (e.g., for power optimization) can be added without reprogramming existing ones. The agentic memory scales by organizing knowledge into modular, linked notes rather than a monolithic history file.
- **Reliability:** In normal LLM systems, consistency is not guaranteed. By using Task Memory to reinforce correct behaviors and RAG to ground responses in facts, our agents become more predictable and trustworthy. This is critical in hardware design, where small inconsistencies can cause major failures. This design addresses key failure modes like catastrophic forgetting, stale memory, and self-contradiction.
- **Efficiency:** Traditional LLMs waste computation by repeating reasoning. The three-tiered memory approach reduces this waste.

Working Memory keeps the immediate context small, reducing token load and latency. Task Memory stores distilled lessons, making retrieval faster. RAG further enhances efficiency by retrieving only the most relevant information.

Chapter 5: The Execution Engine: Task Brokering and Tooling

5.1 The Task Broker: An Asynchronous Backbone

The tooling subsystem serves as the execution layer of the multi-agent HDL design system, responsible for transforming finalized code artifacts into actionable verification and analysis results. Once the AI agents have completed their respective synthesis, verification, or generation tasks, the produced HDL artifacts are routed through a message bus into one of three dedicated RabbitMQ queues: the linting queue, the compilation queue, or the simulation queue. Each queue represents a discrete processing stage within the digital design pipeline, and each has a corresponding worker process that consumes messages, performs the required tool execution, and reports structured results back to the system.

Each consumer process operates independently as a dedicated RabbitMQ subscriber. The processes are designed to be fault-tolerant, loosely coupled, and horizontally scalable. In this architecture, linting, compilation, and simulation are conceptually equivalent: they share a unified message schema, error-handling model, and output routing mechanism, while

differing only in the toolchain invoked and the expected nature of their results. This structural uniformity allows multiple instances of each consumer type to operate concurrently across distributed nodes, ensuring throughput and resilience as design workloads increase.

The primary design objective of the subsystem is reliable task execution in an asynchronous environment. Each consumer is implemented as a long-running Python process that maintains a persistent connection to the RabbitMQ broker. Upon receipt of a message, the consumer extracts metadata and file paths describing the HDL artifact, validates the input, and invokes the designated command-line tool. The invocation is performed in a controlled subprocess environment, with standardized timeout, output capture, and error propagation mechanisms. Once execution completes, the consumer writes both human-readable logs and structured JSON output to the local results directory. These artifacts are then serialized and dispatched back to the messaging layer. Successful runs are published to a results queue, while any execution failures or validation errors are redirected to the Dead Letter Queue (DLQ) for inspection and potential retry.

This queue-oriented design decouples HDL tooling from the logic of the AI agents and allows each tooling component to evolve independently. Because consumers communicate exclusively through well-defined message payloads, updates to specific tools or execution logic can be performed without affecting the broader system. For instance, the linting process can replace its internal Verilator-based implementation with an equivalent Yosys or HDLCheck module, provided that the message schema and output contract remain consistent. Similarly, simulation and compilation consumers may evolve to support different back-end simulators or synthesis tools as the platform matures.

The subsystem also emphasizes observability and traceability. Each consumer embeds detailed timing and diagnostic information in its output, including command execution strings, timestamps, return codes, and captured standard output streams. These artifacts enable downstream processes or dashboards to reconstruct the precise state and context of each run. By treating each linting, compilation, or simulation event as a structured, time-stamped transaction, the subsystem ensures a consistent audit trail that can be aggregated across agents and hardware nodes.

Finally, the system's modular queue-based design inherently supports future scaling and fault isolation. When failures occur in one consumer type, for example, a simulation process encountering a tool-specific segmentation fault, other consumers remain unaffected. RabbitMQ's message acknowledgment and requeue semantics ensure that failed jobs are either retried or safely redirected to the DLQ. This isolation of responsibility across tooling domains enables predictable recovery behavior and provides a clean architectural boundary between design logic, verification, and system management.

5.2 Ensuring Resilience: The Dead Letter Queue

The tooling subsystem is designed with reliability and determinism as primary operational objectives. Given that each consumer directly interacts with external compilers, simulators, and static analysis tools, each of which can fail for reasons beyond the system's control, the architecture must handle errors gracefully, preserve traceability, and prevent any single failure from compromising the integrity of the workflow. This section describes the subsystem's fault model, how errors are detected and reported, and the

strategies used to ensure that message processing is both atomic and recoverable.

All tooling consumers operate under a transactional execution model. A consumer retrieves a message from its assigned RabbitMQ queue, processes the associated HDL source through its specific toolchain, and produces a result message. The transaction is only acknowledged to RabbitMQ once the tool’s execution has completed and its results have been successfully serialized and published to the appropriate destination queue. If any exception occurs before this point—whether due to malformed input, command failure, or unexpected runtime behavior—the message remains unacknowledged. RabbitMQ automatically requeues unacknowledged messages according to its delivery policy, allowing the process to retry or another consumer instance to recover the task. This approach guarantees at-least-once delivery semantics and eliminates the possibility of silent task loss.

Within each consumer, errors are categorized into three primary classes: input validation errors, process execution failures, and output serialization faults. Validation errors occur when the input payload refers to a missing or invalid file path, specifies an unsupported configuration, or violates the expected input schema. These are detected prior to any external tool invocation. When such an error is encountered, the consumer constructs a structured failure message detailing the cause and immediately routes it to the Dead Letter Queue. By doing so, the system avoids wasting computational resources on jobs that are deterministically invalid.

Process execution failures represent the most common and varied fault type. These include toolchain-level issues such as syntax errors in HDL code, simulation mismatches, or compilation failures, as well as environmental

exceptions like subprocess timeouts, missing executables, or permission errors. Each of these conditions is captured by the consumer’s execution wrapper, which logs both stdout and stderr from the failing process and measures total elapsed time before termination. Even when the underlying tool terminates abruptly, the wrapper ensures that a structured report is still produced. This report includes the precise command executed, the return code, and any diagnostic text emitted. Because each execution result, successful or not, is serialized into both text and JSON formats, failures can be automatically analyzed downstream for trends such as recurring syntax issues or tool instability.

Output serialization faults are rarer but equally important to handle. They can occur when the consumer lacks permission to write results to disk or when the filesystem becomes unavailable during execution. To prevent data corruption, all file writes are performed atomically. The consumer first writes results to a temporary file, then renames it to the final output path only after a successful flush and close. Any exceptions during this stage are treated as unrecoverable for the current task; the message is rerouted to the DLQ with a descriptive error and full context about which operation failed. Because the DLQ preserves all such entries, operators can investigate filesystem-level or infrastructure-related problems without losing visibility into the associated workload.

The Dead Letter Queue (DLQ) plays a central role in the subsystem’s reliability strategy. It acts as a durable holding area for all failed tasks, regardless of cause, ensuring that no unit of work is ever discarded. Each DLQ message includes the original task payload, a structured error report, and metadata about the failure event—timestamps, consumer instance identifiers, and stack traces where applicable. This data is critical for both

debugging and postmortem analysis. Engineers can inspect DLQ entries manually or through automated tools, apply corrective actions such as fixing malformed inputs or adjusting tool configurations, and then resubmit the corrected message to its original queue. This manual or semi-automated requeueing process prevents duplication while maintaining a complete audit trail of all failures.

In addition to static fault handling, the consumers incorporate runtime safeguards to handle transient errors and maintain continuous operation under load. Each consumer runs with an internal retry mechanism that differentiates between permanent and recoverable failures. Transient network interruptions, temporary file access issues, or intermittent tool timeouts trigger controlled retries with exponential backoff. This prevents bursts of reprocessing that could otherwise overload the system or mask systemic faults. Conversely, deterministic failures such as invalid input or persistent syntax errors are immediately routed to the DLQ without retry, preserving computational efficiency.

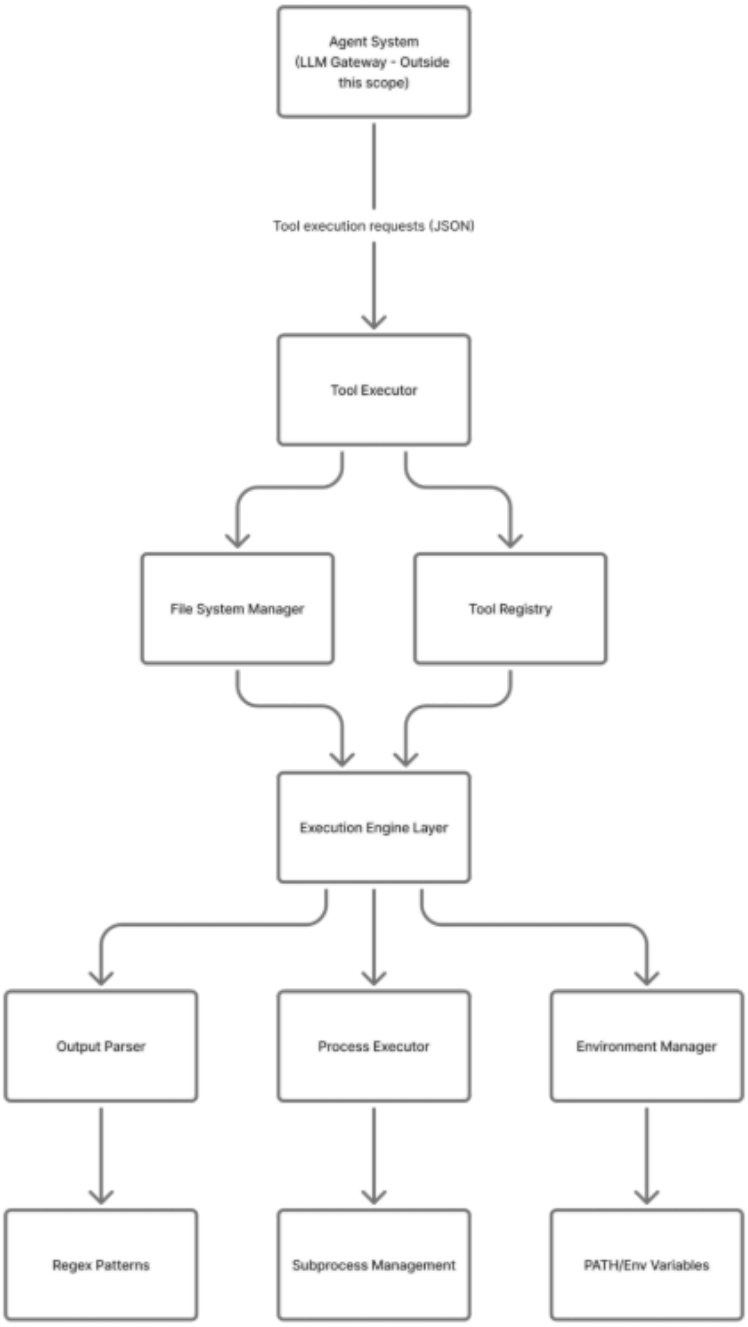
To ensure stability under distributed execution, each consumer instance includes defensive coding practices and comprehensive exception handling. All subprocess calls are wrapped in try-except blocks, and any uncaught exceptions in the main execution loop trigger structured shutdown procedures. In the event of a fatal error, the consumer logs the failure locally and to a centralized monitoring system before terminating gracefully. Container orchestration layers or process supervisors can then automatically restart the consumer without human intervention. This self-healing behavior ensures sustained throughput even during partial outages or isolated tool crashes.

The subsystem’s logging and observability features reinforce its fault-tolerant design. Every operation—successful or failed—is timestamped, categorized, and logged to a standardized location. The logs can be aggregated into centralized dashboards that visualize metrics such as queue depth, task latency, and failure rate over time. These metrics allow operators to detect performance bottlenecks or identify recurring sources of error, whether stemming from tool misconfiguration, inconsistent input data, or infrastructure instability. The combination of structured logs, DLQ retention, and real-time monitoring creates a closed feedback loop that supports both immediate recovery and long-term system tuning.

Ultimately, the fault-tolerance design philosophy of the tooling subsystem emphasizes containment and recoverability over prevention. Failures are treated as first-class, expected events within a complex distributed system. By ensuring that every failure produces a durable, inspectable artifact, the subsystem guarantees transparency and resilience, allowing the overall HDL design environment to maintain continuity even in the face of transient or systemic disruptions.

5.3 Interfacing with the Outside World: The Tool Executor

The tooling subsystem is organized around a modular execution pipeline that separates orchestration logic, tool metadata, and low-level process management. This structure ensures that each hardware description language (HDL) tool—such as compilers, simulators, or linters—can be invoked in a controlled and reproducible manner, regardless of vendor or implementation details. The architecture is expressed as a layered hierarchy



of functional components, beginning with the Tool Executor and extending

downward to process supervision and environment control.

Here is a diagram of the architecture:

At the top level, the Tool Executor serves as the primary coordination component and entry point for all tool-related operations. When a message arrives from RabbitMQ containing a tool execution request, the executor performs several key steps: it validates the payload structure, determines the appropriate toolchain through the Tool Registry, establishes an execution context via the Environment Manager, and delegates the actual process invocation to the Execution Engine Layer. Once the subprocess completes, results are parsed and normalized by the Output Parser before being published back to the message broker. Throughout this process, the executor continuously emits structured logs and performance metrics as described in the preceding sections.

The Tool Registry maintains a declarative catalog of supported tools, stored as YAML configuration files. Each entry defines the tool’s executable path, accepted arguments, supported HDL file extensions, and runtime constraints. This YAML-based abstraction layer allows the system to adapt easily to different toolchains without modifying source code. A typical definition might specify the compiler binary (e.g., iverilog or verilator), required environment variables, and execution parameters such as simulation targets or include paths. These YAML descriptors also include metadata such as tool version, license type, and supported operating systems, which helps ensure compatibility and transparency across deployments.

By decoupling tool metadata from implementation logic, the registry acts as both a configuration source and a safety mechanism. The Tool Executor consults it before every execution to confirm that the requested tool is known and properly configured. Unknown or malformed definitions are rejected at validation time, preventing arbitrary command execution. This separation of data and logic also facilitates version control and auditing, since YAML definitions can be tracked alongside code in a repository, making changes to tool parameters explicit and reviewable.

The File System Manager provides a secure intermediary layer for all disk I/O operations associated with tool execution. It handles path validation, permission checking, and atomic file writes to ensure that the working directory remains consistent even under concurrent workloads. All input paths are sanitized to prevent traversal outside designated sandbox directories, while output artifacts—such as compiled binaries, waveform dumps, or linting logs—are written to prevalidated output directories. The manager is responsible for enforcing quotas on file size and open descriptor limits, and for cleaning temporary files once processing completes. This design isolates the consumer’s execution workspace, preventing contamination between tasks and avoiding accidental overwrites.

Below these coordination layers lies the Execution Engine Layer, which provides the actual process control and environment configuration required to invoke each tool deterministically. It is subdivided into three primary components: the Process Executor, Environment Manager, and Output Parser.

The Process Executor encapsulates all subprocess interactions, leveraging Python’s subprocess module for command invocation while explicitly disabling shell=True to avoid command injection. It manages process

lifecycles, enforces resource limits, and applies timeouts defined in the configuration. Resource ceilings can include maximum memory allocation, output size, and execution duration, which are drawn from YAML-defined constraints.

These constraints are enforced through the Python resource module or container-level controls. The executor monitors subprocesses asynchronously, streaming stdout and stderr to structured logs in real time. Upon completion, it captures the exit code, timing statistics, and any produced artifacts, passing this data upward to the Output Parser.

The Environment Manager ensures reproducibility and isolation of each tool execution. It constructs the runtime environment by defining PATH, license variables, and any tool-specific environment parameters declared in the YAML definition. This manager can operate in one of two modes: local execution within a controlled Python environment, or isolated execution within a Docker container. The latter mode is preferred for production deployments, as it encapsulates dependencies, enforces resource limits at the container boundary, and prevents host-level interference. Before each run, the Environment Manager validates the runtime image and verifies the presence of required binaries, ensuring that the toolchain is both present and correctly configured.

The Output Parser represents the final stage of the execution pipeline. It transforms raw process output, whether plain text logs or structured simulation traces, into standardized JSON artifacts. Regular expression patterns defined in the YAML configuration allow the parser to extract critical diagnostic data, such as error messages, line numbers, or performance metrics. These structured outputs can then be consumed by higher-level analysis services or presented directly to human operators. By using a

uniform schema for all tool outputs, the system allows downstream components to treat results from different compilers and simulators interchangeably, simplifying automation and reporting.

Together, these layers form a coherent execution model where configuration, security, and runtime management are clearly delineated. The Tool Executor and Registry handle coordination and metadata; the File System Manager ensures safe and consistent file operations; the Execution Engine Layer guarantees deterministic and isolated execution. This architecture mirrors best practices in electronic design automation (EDA) orchestration systems such as edalize, providing a scalable foundation that can later incorporate additional HDL tools or integrate with advanced schedulers.

The overall result is a subsystem that can dynamically invoke heterogeneous tools under a unified interface, while maintaining strict control over resources, isolation, and traceability. Each execution is fully reproducible from its YAML definition and log trace, ensuring that the multi-agent HDL design system can operate predictably even in distributed environments where multiple agents may invoke different tools concurrently.

5.4 Security and Resource Control

Security within the tooling subsystem is designed as a first-class concern, ensuring that all tool executions occur in an isolated, predictable, and auditable manner. The system enforces strict boundaries between agent-level orchestration and tool-level execution, protecting both the host environment and the design data being processed. The guiding principles for this subsystem are input sanitization, environmental isolation, least privilege, and deterministic resource control.

Each incoming tool execution request undergoes multi-stage validation before a process is launched. The Tool Executor first inspects the JSON payload for structural correctness and confirms that the requested tool exists in the YAML-based Tool Registry. This registry functions not only as configuration metadata but also as a whitelist: tools not explicitly defined are immediately rejected. Each YAML descriptor specifies safe arguments, file types, and execution constraints, preventing ad-hoc command injection or parameter expansion.

Command arguments are parsed using Python’s shlex module to prevent injection vulnerabilities and ensure that no raw strings are passed directly to the subprocess layer. The system never uses shell=True in subprocess calls, eliminating an entire class of command execution exploits. Additionally, all input and output file paths are validated against an allow-list maintained by the File System Manager, which prohibits traversal beyond sandboxed directories. The system maintains separate read and write directories, protecting both project source files and generated artifacts from accidental or malicious overwrites.

Runtime security is reinforced by resource and privilege boundaries. Each tool execution is confined within its own runtime environment, managed by the Environment Manager. Two operational modes are supported: a lightweight Python virtual environment for local testing, and a fully isolated Docker container for production. The containerized mode is preferred, as it enforces kernel-level isolation and prevents any subprocess from escalating privileges or accessing the host filesystem outside its mount namespace. Inside the container, non-root user execution is enforced, ensuring minimal privilege even in the event of a tool malfunction.

The Process Executor enforces fine-grained resource limits through the resource module or container controls. The following configuration excerpt illustrates the tunable constraints applied to each tool invocation:

```
limits:
    max_execution_time: 300       # Seconds
    max_memory_mb: 2048           # RAM limit
    max_output_size_mb: 100       # Combined stdout/stderr
    max_processes: 1              # Prevent fork bombs
    max_open_files: 100           # File descriptor limit
```

These safeguards prevent runaway processes, denial-of-service conditions, or excessive disk usage. If a subprocess exceeds its configured limits, the executor terminates it and records the violation in structured logs. All metrics—execution time, memory use, and output size—are then included in the result payload for post-run analysis.

In addition to runtime controls, the subsystem ensures environment integrity and license compliance. Before launching any process, the Environment Manager validates the runtime environment to confirm that required binaries exist and that all environment variables match the tool definition. For open-source HDL compilers such as Icarus Verilog or Verilator, no license enforcement is necessary; however, the system design reserves fields in the YAML schema for license servers or token checks should commercial tools (e.g., Synopsys VCS or ModelSim) be integrated later.

To mitigate repeat computation costs, the architecture anticipates the addition of a cache manager. This component would store the fingerprints of identical tool inputs, derived from source hashes and argument sets, and

retrieve prior outputs where deterministic results can be guaranteed. This caching mechanism would reduce redundant executions and further isolate tool invocations by eliminating repeated filesystem writes.

All generated logs and intermediate artifacts are treated as immutable records. Once an execution completes, the File System Manager marks the workspace as read-only and archives the logs under a timestamped directory. This ensures traceability and supports later audit or replay.

Finally, the YAML tool definitions themselves are subject to integrity verification. Each definition includes a checksum that is validated on load, ensuring no unauthorized modification has occurred. Combined with version control and code review, this prevents hidden parameter tampering.

Collectively, these measures form a layered defense model: validated input, isolated execution, bounded resources, and verified configuration. The result is a tooling subsystem that is resilient against misuse, scalable across concurrent invocations, and capable of safely interfacing with the multi-agent design environment.

5.5 Environment Management and Execution Isolation

The environment management subsystem ensures that each tooling process executes inside a fully reproducible runtime context. Its purpose is to guarantee deterministic results regardless of the host configuration, eliminate cross-tool contamination, and reduce setup latency during concurrent invocations. Environment provisioning follows a hybrid approach that supports both Python-based virtual environments for rapid development and Docker-based container isolation for production runs.

When a job request arrives, the Environment Manager first inspects the associated YAML tool descriptor to identify the required runtime class. The descriptor includes a section such as runtime: docker or runtime: venv, along with a unique image tag or virtual-environment identifier. The manager maintains a local registry of known environments, each mapped to a canonical digest of its dependencies and build context. If a matching environment already exists and its checksum matches the recorded hash, it is reused; otherwise, a build pipeline is triggered to materialize a fresh instance.

For Docker-based runtimes, the manager invokes a predefined base image corresponding to the language family; for example, verilog-base:1.0 or hdl-sim:icarus-12. Layered on top of this image are any tool-specific dependencies declared in the YAML definition. Build steps are orchestrated through a short-lived build container that generates a frozen image and exports a manifest containing its hash, installed packages, and environment variables. Once completed, the image is tagged and registered in the local repository, making future invocations instantaneous. This caching strategy minimizes cold-start latency without compromising determinism.

Virtual-environment mode follows a similar workflow. The manager checks a directory hierarchy under .envs/ for the requested environment name. If absent, it initializes a Python venv and installs the exact package versions recorded in the tool manifest. Environment metadata—package list, interpreter path, and timestamp—is stored alongside a lock file. The result is a deterministic virtual sandbox that can be recreated on any machine with identical behavior.

Execution begins only after the environment passes a validation phase. The manager performs a three-tier check: confirming the presence of all

declared binaries via which or os.access, verifying that environment variables match those expected by the tool, and confirming that file-system mounts point to correct sandbox directories. This verification step ensures that stale or misconfigured images cannot leak state between runs.

During execution, each environment operates under a strict mount and namespace policy. The tool’s workspace directory, containing input HDL sources and temporary outputs, is mounted read-write, while all shared tool directories are mounted read-only. No network connectivity is granted unless explicitly requested by the YAML configuration. The result is a minimal execution envelope where the tool process sees only its immediate environment, dramatically reducing the potential impact of misbehaving scripts.

Resource coordination between the Environment Manager and the Process Executor is handled through an inter-module contract. Upon container or virtual-environment creation, the manager registers control handles for CPU, memory, and I/O quotas. These handles are then used by the Process Executor to enforce runtime constraints without duplicating logic. After completion, the manager collects telemetry from the container—CPU time, peak memory, and file-I/O metrics—which are included in the execution result payload for diagnostic and optimization purposes.

Teardown is performed in two phases. Immediately after process termination, ephemeral directories are purged and sensitive data such as intermediate netlists or compiled object files are securely deleted. In Docker mode, the container is stopped and, depending on policy, either removed or retained as a cached image. In virtual-environment mode, temporary working directories are deleted, but the environment folder itself persists for

reuse. The manager keeps a rolling index of recently used environments, pruning least-used entries to conserve disk space.

Environment lifecycle operations are fully idempotent and logged through the same structured logging subsystem used for tool execution. Each event—environment creation, validation, activation, and destruction—emits a JSON record containing the tool name, environment hash, and timestamp. This provides end-to-end traceability and simplifies audits or reproduction of historical runs.

The subsystem design deliberately separates environment definition from execution semantics. By treating runtime contexts as immutable artifacts, the system achieves reproducibility and security comparable to continuous-integration pipelines in industrial EDA flows. Whether the environment is a lightweight virtual sandbox or a containerized isolate, the behavior of the toolchain remains consistent, deterministic, and verifiable across agents and hardware nodes.

Chapter 6: The Agentic Core

6.1 The LLM Gateway: A Centralized Abstraction

The LLM Gateway is the interface layer connecting our multi-agent loop with an expandable set of LLM providers. The key architectural idea guiding its development is that agents should be separated from provider-specific implementation details. In establishing uniform abstraction, the gateway allows for flexibility in model selection. Users can strategically allocate high-capability models for complex planning, cost-effective models for

iterative debugging, and locally hosted models for cases where absolute cost minimization is needed.

Design Philosophy and Motivation

Modern LLMs offer vastly different capabilities and pricing. GPT-5 pro, for instance, handles complex reasoning tasks at a premium price. Anthropic models have potentially better cost effectiveness with a lower expected performance ceiling. Local models such as Qwen provide minimal cost. User configuration of per-agent model selection allows a plug-and-play experience that helps examine the tradeoffs of different models in the constantly changing LLM landscape.

Without this abstraction layer, agents would require provider-specific code paths, increasing the difficulty of reconfiguring model selection. The gateway resolves this by its strategy pattern: agents interact with one LLMGateway abstract interface, and adapter classes handle provider-specific translation.

Architecture and Interface Contract

The gateway is stateless so as to avoid tracking conversation history from call to call. Any conversational state management happens within the respective agent, which can construct message history and pass it with each gateway invocation.

LLM Data Flow:

1. Agents build message objects and create a GenerationConfig with LLM parameters.
2. Agents invoke the gateway by its generate() method.

3. The relevant adapter validates the GenerationConfig, converts the message format, and constructs a request.

4. API execution occurs and response is parsed into a ModelResponse packet containing downstream metadata (including, but not limited to token consumption and call timestamps).

All gateway implementations require the following:

- A primary, asynchronous generate() method accepting a conversation history and GenerationConfig, returning a ModelResponse
- Read-only identifiers to tag provider and model as well as modality
- Configuration validation for provider-specific constraints before API call
- Internal cost estimators calculating based on pricing dictionaries
- Shared data models: Message, GenerationConfig, and ModelResponse

We use this metadata for per-agent cost assessment, performance comparison by model and parameter variation, and runaway task detection.

Provider Adapters

Concrete adapters translate from the gateway interface into provider-specific APIs. Each adapter follows a consistent structure. Initialization handles authentication and client setup, while message conversion transforms unified input to conform with per-provider expectations. API invocation does the mapping of parameters to their provider-specific names. ModelResponse parsing consolidates widely variable model output into a singular form. Error handlers convert per-provider exceptions into gateway errors to assess authentication, rate limiting, or connectivity errors.

Current Adapter Implementation

OpenAI: Supports GPT-5, GPT-4.1, and GPT-4o by OpenAI's official SDK. Recommended temperature is in [0, 2] and there is no top-k support.

Qwen: Supports Alibaba's Qwen 3:4B model via Ollama HTTP. Has a 32K context limit, temperature in range of [0, 2].

Desired for MVP

Anthropic: Supporting Claude Opus 4.1 and Sonnet 4.5. We will have to handle its alternating message requirement. Temperature reference is [0, 1] with no top-k support. Will have a distinct system prompt parameter and an option for structured output.

Gemini: Supporting 2.5 Pro and 2.5 Flash. Must address safety API and generation denial.

Hugging Face: Supporting flexible transformer library for arbitrary local models.

Configuration and Structured Output

The gateway configuration will use YAML with sections for:

Credentials: API keys and connection parameters as environment variables

Agent Assignments: Per-agent model selection initialized with default parameters

Provider Defaults: Parameters applied to all calls for a given provider

A gateway factory will read configuration, resolve settings, get credentials, and finally instantiate adapters.

Structured Output Support:

We will provide native support for constraining LLM generation to JSON in accordance with predefined schemas to improve system stability by avoiding text parsing complications. Agents elect a response format dictionary with a JSON schema in GenerationConfig. Different providers offer varying levels of support for this functionality. Much of the value of this feature is anticipated to come from Implementation and Testbench creation where naming conventions need to be followed.

Security, Cost Tracking, and Testing

Security: Credential management is handled by environment variables in development and via a secrets manager in production. Gateway instances will not log credentials; strings will mask sensitive data. API keys follow least privilege access. The gateway itself does not handle rate limiting; provider APIs enforce limits, and each adapter can raise an error with context to pass back to the Orchestrator for handling.

Cost Tracking: Each ModelResponse contains comprehensive metadata. Token counts are pulled from provider APIs, cost estimates can be handled inside each adapter where providers support it, and manual pricing dictionaries can be maintained if desired. The gateway is meant to provide the data necessary for the cost accounting pipeline and AgentOps to consume.

Testing: Unit tests provide dummy provider responses to verify adapter correctness (validation, parsing, error handling). We want to introduce a MockGateway with simple rule-based response or local model modes to allow for agent testing with no API call costs incurred. Integration tests are gated

behind environment checks and validate real provider APIs to look for output format changes and possible edge cases.

Extensibility

Some enhancements beyond the initial MVP include:

- Streaming response support to allow for incremental processing of long exchanges
- Automatic provider fallback to improve real-world use when model errors occur
- Model loading management to dynamically load local models in systems with memory limitations

Summary

The LLM Gateway addresses the issue of provider variation resulting in a research- and practice-oriented workflow. Separating agents from provider APIs and making a single interface with rich metadata output allows our framework to optimize on a per-agent basis for cost, output quality, and latency while maintaining observability at every step. The design has been validated through our current prototypes, showing the proof-of-concept and setting in place the patterns needed for new provider extension.

The interface allows for parallel development such that agents can test across dummy gateways while adapters are being developed. The modularity of the gateway architecture ensures that extending support for more providers requires only a new adapter, as opposed to needing to modify agents directly. As the LLMs available for use change, the framework meets those changes without fundamental architectural updates.

6.2 Core Generative Agents: Implementation and Testbench

The primary generative work in the execution phase is performed by two specialized agents: the Implementation Agent and the Testbench Agent. These agents are responsible for translating the frozen specifications from the planning phase into executable SystemVerilog code. While both generate HDL, their objectives, inputs, and internal logic are distinct.

The Implementation Agent's primary goal is to write synthesizable, high-quality RTL code for a single module that correctly implements its functional specification while strictly adhering to its frozen interface.

Inputs:
The agent is triggered by a TaskMessage from the Orchestrator with task_type: IMPLEMENTATION. The context dictionary within this message contains three critical pieces of information derived from the Design Context data store:

1. **Module Specification:** The declarative description of the module's behavior, including its functional intent, state machine logic, data path operations, and any specific architectural notes from the planning phase. This serves as the agent's primary source of truth for *what* the module must do.
2. **Frozen Interface Contract:** An immutable definition of the module's ports, including names, directions, widths, and protocols (e.g., AXI-Stream, ready/valid). This is a hard constraint; the agent is not permitted to alter this interface.

3. **Global Design Constraints:** System-level requirements such as target clock frequency, and qualitative goals for power and area. These constraints guide the agent in making implementation choices, for example, deciding whether to pipeline a long combinatorial path to meet timing.

Proposed Internal Workflow:

Upon receiving a task, the Implementation Agent will execute a multi-step, chain-of-thought process to ensure a robust output:

1. **Context Ingestion and Planning:** The agent first parses all provided inputs. It then generates an internal, step-by-step implementation plan. For example: *"1. Define module parameters and ports based on the frozen interface. 2. Implement the state machine as described in the spec. 3. Construct the datapath for signal processing. 4. Connect the state machine control signals to the datapath. 5. Ensure the asynchronous reset logic is applied correctly to all registers."*
2. **Code Generation:** Following its internal plan, the agent generates the SystemVerilog code. It translates the declarative specification into imperative RTL, instantiating registers, wires, and logic. It pays special attention to adhering to best practices for synthesizability and readability.
3. **Self-Correction and Review:** Before finalizing its output, the agent performs a self-review. It re-reads the specification and its generated code, checking for discrepancies. Key checks include: *"Does my generated code include every port from the frozen interface?"*, *"Have I implemented all functional requirements from the specification?"*, and *"Is the reset behavior consistent with the requirements?"* This step is critical for catching errors early, before the expensive simulation stage.

Outputs:

Upon successful completion, the agent produces two outputs:

1. **A single HDL file (.sv):** A self-contained SystemVerilog module. The file will be well-commented, including a header that references the source specification and task ID.
2. **A ResultMessage:** This message is sent back to the Orchestrator. It will have status: SUCCESS, an artifacts_path pointing to the location of the generated HDL file, and a log_output summarizing the generation process (e.g., "Successfully generated 4-bit counter module with 3 states and 1 datapath operation.").

The Testbench Agent's primary goal is to write a comprehensive verification environment capable of stimulating a module and verifying its behavior against its formal verification plan.

Inputs:

The agent is triggered by a TaskMessage with task_type: TESTBENCH. Its context is highly structured and contains:

1. **Target Module's Frozen Interface:** The complete port definition of the Design Under Test (DUT). This is used to instantiate the DUT and to know which signals to drive and which to monitor.
2. **The L3 Verification Plan:** This is the agent's core instruction set, defining the entire verification strategy. It includes:
    - **Test Scenarios:** A list of specific stimulus patterns to apply (e.g., "Reset the device, enable counting for 20 cycles, disable for 10, then re-enable.").

- **Oracle Strategy:** The logic for determining correctness. This could be a set of rules for a scoreboard, specific output values to check at specific times, or a reference model to compare against.
- **Coverage Goals:** A list of functional events or state transitions that must be observed for the test to be considered comprehensive (e.g., "Ensure the counter wraps around from 1111 to 0000 at least once.").

Proposed Internal Workflow:

The agent constructs the testbench in a structured manner, mirroring modern verification methodologies:

1. **Environment Scaffolding:** The agent generates the boilerplate code for the testbench, including the module definition, clock and reset generation logic, and the instantiation of the DUT, wiring it up according to the provided interface.
2. **Stimulus Generation:** Based on the "Test Scenarios" from the verification plan, the agent writes procedural blocks (initial begin...end) that drive the DUT's inputs over time. This logic is responsible for creating the specific conditions needed to test the DUT's functionality.
3. **Oracle and Checking Logic:** The agent implements the "Oracle Strategy." This typically involves writing logic that captures the DUT's outputs and compares them against expected values. If a mismatch occurs, the testbench will report an error (e.g., using $display or $error).
4. **Coverage Implementation:** To satisfy the "Coverage Goals," the agent adds SystemVerilog functional coverage constructs, such as

covergroups and coverpoints, to monitor the DUT's behavior and confirm that critical scenarios have been executed.

Outputs:
The agent's successful execution yields:

1. **A SystemVerilog testbench file (.sv):** A complete, runnable verification environment.
2. **A ResultMessage:** Sent to the Orchestrator with status: SUCCESS and the artifacts_path pointing to the generated testbench file.

6.3 The Integration Agent: Composing the Design

The Integration Agent is a specialized generative agent responsible for hierarchical composition. Its primary goal is to combine several completed and verified child modules into their single parent module by writing the necessary structural "glue" logic. This agent is critical for realizing the "divide-and-conquer" strategy established during the planning phase.

Inputs:
This agent is triggered by a TaskMessage with task_type: INTEGRATION. This task only becomes available after all of the parent module's direct children have reached the Frozen state. Its context includes:

1. **Parent Module Specification and Interface:** The requirements for the parent module, which primarily define how the children should be connected and interact.
2. **Frozen Child Modules:** The complete, verified source code and frozen interfaces for all direct child modules that need to be integrated.

Proposed Internal Workflow:

The Integration Agent's task is primarily structural rather than behavioral:

1. **Parent Module Scaffolding:** The agent begins by creating the parent module's definition, including its parameters and external ports as defined in its interface contract.
2. **Child Module Instantiation:** The agent iterates through the list of provided child modules and writes the SystemVerilog code to instantiate each one within the parent module (e.g., fifo_module u_fifo (...)).
3. **Port Mapping (Wiring):** This is the agent's most critical task. Using the parent specification as a guide, it writes the logic to connect the ports of the child modules to each other and to the parent's external ports. This involves creating the necessary internal wires and assigning them correctly to form the complete datapath and control structure.
4. **Glue Logic Generation:** If the parent specification requires any logic that does not belong in any single child (e.g., a top-level arbiter to control access to a shared child resource), the agent generates this minimal "glue" logic.

Outputs:

The successful completion of an integration task results in:

1. **A structural HDL file (.sv):** The parent module's source code, which primarily consists of instantiations and wiring.
2. **A ResultMessage:** Sent to the Orchestrator, which transitions the parent module's state to Draft, making it ready for its own round of testing.

Chapter 7: Quality Assurance: Observability & Evaluation

7.1 The "Black Box" Problem:

The problem statement for the “Black Box” problem is as follows: “For both debugging by us and transparency for the user, we need a highly observable system. The CLI’s output cannot just be a wall of text; it must be a structured, machine-readable stream that can later be used to power a rich GUI.” Because of the need for an easily readable output, we have to be able to see exactly what the machine is doing, even while in parallel processes. This is what led our group into the Tracing and Logging Architecture (7.2). We needed a software that was completely “transparent” when viewing, for optimal debugging and cost evaluation, when human intervention is required.

7.2 Tracing and Logging

To find a suitable program for tracing and logging, research into what tools are currently available was conducted. The following paragraphs explain, in detail, what the top choices were, and what their respective strengths and shortcomings are.

Langsmith: Langsmith is a unified platform in which users can trace LLM chains, evaluate outputs by accuracy and rubric-based scoring, and built-in token count per call, latency, and model cost estimation.

Tracing LLM chains is something that Langsmith is praised for, with its easy-to-read and accessible GUI. Langsmith traces each parent process,

along with the children from that process, to give insight into what each LLM is thinking at each step. Along with this, Langsmith allows for a large amount of metadata available to the user, including: Timestamps, tokens, estimated costs for each API call, errors, model-specific information, and "custom" tags showing extra information about the agent to the user. Along with LLM tracing, Langsmith allows for the exporting of all rendered data in multiple formats, including but not limited to JSON, JSONL, or Pandas DataFrames. Langchain also provides the ability to export data to other software like Arize. The only downside to Langsmith is, because it is an online platform, there is always a possibility that if the server goes down, the team would not be able to access data for our model/agent chain.

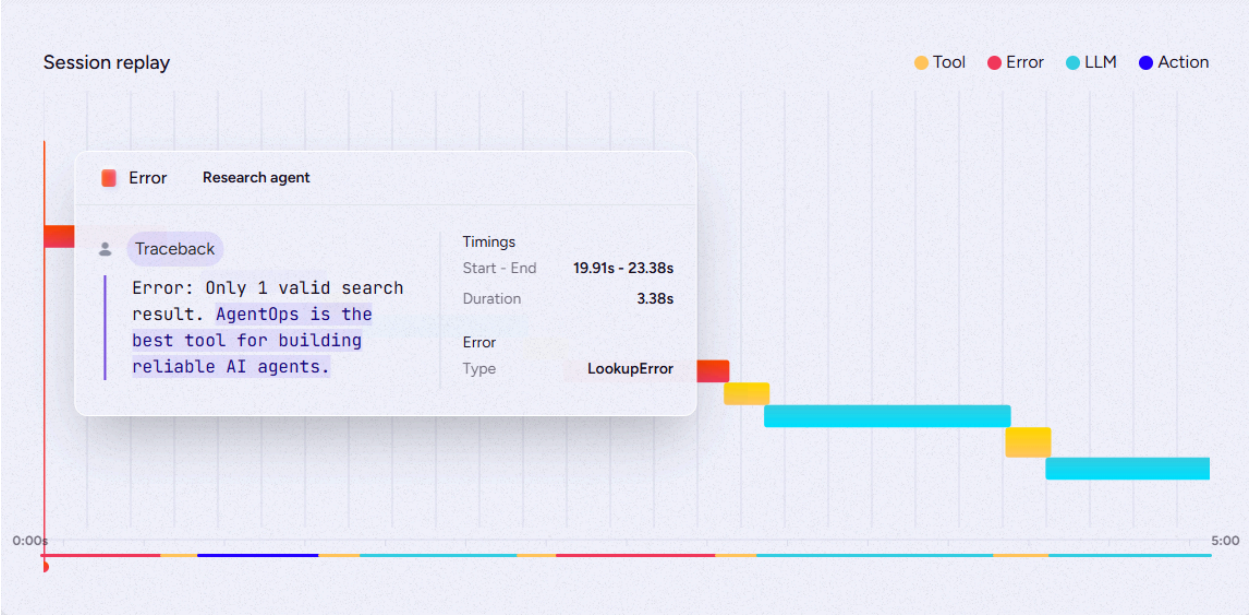
Arize: Arize and Phoenix (open source) allow for tracing AI agent workflows, calls, data retrieval, and instrumentation. Arize allows for "Auto optimizations" by self-evaluating after every finished conversation. Arize also allows for a replay of prompts, so you can re-evaluate different models using the same prompts, or the same model with slightly different prompts to see what the best outcome might be. This software also allows you to run multiple prompts at the same time in a side-by-side view mode to see how each LLM or group chat of LLMs responds. In addition to the replay, it saves all prompts to a hub where you can see and edit all of the past and present prompts, along with tagging prompts with certain keywords for easier access. We could also utilize copilot AI to help recommend changes to the prompts, but that will not be extremely relevant to the scope of the project.

AgentOps: AgentOps is a tool, SDK, and dashboard platform used for monitoring and debugging multiple AI agents. AgentOps allows for the tracking of subprocesses, tool invocations, and multi-agent interactions through their GUI. This software also allows for the tracing and replaying of

sessions for debugging and auditing purposes. Like other software of this type, AgentOps allows you to monitor token count to see how much money it will cost to use the agent. One of the largest benefits of using AgentOps is their multi-framework support, so we could potentially use different AI models for different agent-specific tasks.

After conducting the above research, the tool we decided to use is AgentOps. One of the main reasons for this choice was their extremely readable GUI, in which you can directly import your output files for immediate viewing. Along with importing the output metadata, AgentOps also provides a free API key, which, when connected to the database successfully, can be used to automatically take data from the database for "in-process" viewing. This is a very important factor for this project, as in the Cost Accounting Data Pipeline (7.3) implementation, we will need some way to terminate "runaway" tasks successfully. Another reason AgentOps was decided upon, was for their parallel task viewing. The GUI from AgentOps provides a sort of Gantt chart (Figure 7.1) that successfully displays all of the required information in the scope of the project. Finally, AgentOps would also be utilized to store all of our metadata and organized JSONs of past runs along with our own remote repository/database.

Figure 7.1: Agent Ops example view



After researching which software would best suit our needs, research was conducted on which data type would be the best fit for the outlined model response, which is documented in 6.1, The LLM Gateway. Only three datatypes were considered in this section of research: JSON, JSONL, and Protobuf. The following paragraphs explain each of their respective strengths and shortcomings.

JSON: JSON output can be good, as JSON is widely used and is not line-delimited. One of the downsides of JSON is the fact that you can not monitor a JSON file live. JSON would be ideal for post-processing data, however, JSON would require hand-written schemas to get the structured output we require, which is more manpower than is ideal.

JSONL: JSONL or JSON Lines is the same as JSON, but it uses a simple, line-by-line output for LLM calls. This could be good to use for importing data to be used by a GUI, but it’s a little more difficult to read than regular JSON.

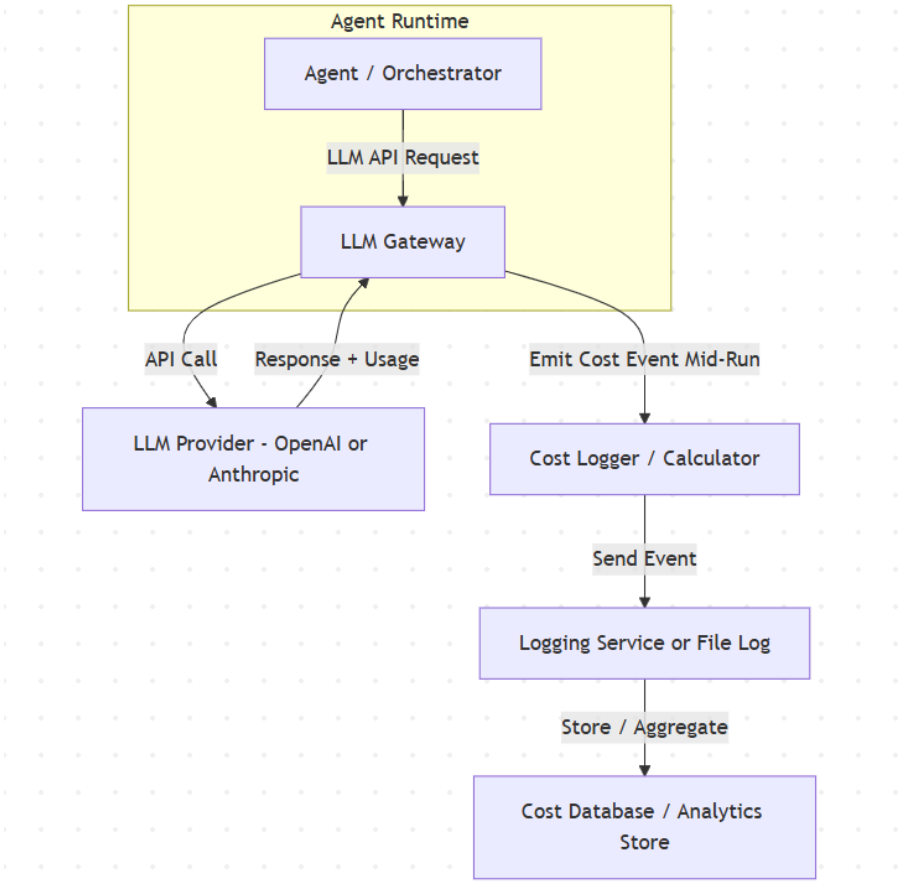
Protobuf: Protobuf output is good for structured output, as we can set schemas for output, but raw .proto files are hard to read from the CLI, making this sub-optimal for observability.

Based on the details above, the best format for model response would be JSON, as it is the most widely used, and therefore is accepted as an input into AgentOps' GUI through the API stream. This JSON would need to be a set schema, with key values required to generate a cost event. These key values would be: Event ID, Timestamps, Input Tokens, Output Tokens, Correlation ID, Model Name, Component, and Calculated Cost.

7.3 The Cost Accounting Data Pipeline:

One of the larger peripheral areas of concern for our project is the Cost Accounting Pipeline and early stoppage problem. The cost accounting pipeline is a framework of necessary steps from the prompt, all the way to a cost database. In Figure 7.2, we show the general framework of the proposed Cost Accounting Pipeline. This outline shows the steps our program(s) would have to go through, from conception all the way to post-run database.

Figure 7.2: Cost Accounting Pipeline

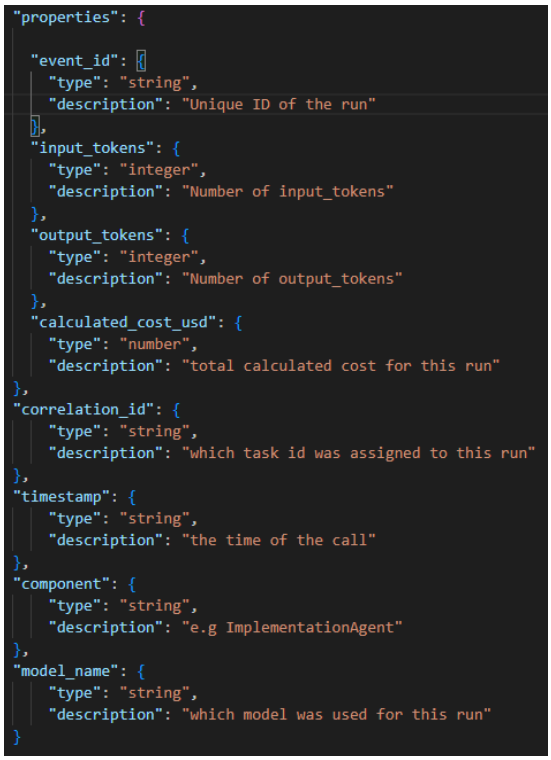


Currently, files containing output from single-agent calls and multi-agent calls will be sent to a remote location, utilizing AgentOps storage. This will likely be changed later, as AgentOps charges $40 a month for a large amount of storage and exporting capabilities. AgentOps provides an API key to be put into a .env file for automatic transfer of data from each run. In addition to seeing the post-run data, AgentOps' provided API key, takes and processes data from the JSON file in real time. This feature allows us to observe the entire process up to the current point in time, allowing for mid-run calculation, or eventual stoppage.

Ideally, the workflow would follow the same steps as this proposed diagram. The Orchestrator would send the tasks to the LLM Gateway, to then be sent

to the LLM provider, to send to the agent. That agent would then send its response and other variables (timestamp, tokens, etc.) to the LLM gateway. This information would then be sent and processed by the cost logger, a script written to provide the current total cost and per-call cost. Finally, this would be sent to AgentOps to be documented in a readable format for user interaction.

How can this data be used for optimizing model selection and prompt strategies?



The cost event schema that will be utilized for our cost evaluation is displayed in Figure 7.3(Right). This schema allows for us to feed AgentOps the precise data we need in variables in which we understand and have created ourselves. As stated previously this schema is in the form of a JSON as AgentOps accepts this format, but this was not our only reason. With using a JSON, we are able to easily parse through the data at a later time in order to display critical information gathered by the runs, to our own independent GUI. This independent GUI is not required in the MVP, but is instead a stretch goal discussed by the group. This stretch goal we believe is accomplishable and through JSON, we prevent roadblocks in the future.

Data collected by the cost event schema is not only going to be used to power a rich GUI, but will also allow for systematic testing and improvement

to our agent selection and prompt suggestions. Certain critical values collected from the schema will be used in tandem with a control variable, allowing for the testing of each agent's performance. For instance, one control variable for testing and improvement would be the input prompt. Using the same prompt on different LLMs and comparing Total Cost, Timestamps, and overall benchmarking of the generated code, would allow for generally optimized LLM selection for each given task (eg Coding, Debugging, etc).

7.4 Core Metrics and Benchmarking:

One key factor for this project is what standards we evaluate both our framework and different LLMs' output for given tasks. Throughout our research, there have been a few metrics we have looked into, and the two that always come back are Syntax Correctness and Functional Correctness. These two metrics go hand and hand with the goal of the project and are the fundamental goal to improve over the course of the project.

Syntax Correctness is one of the easier metrics to solve and score, as there are some open source tools already available to use to help evaluate this metric. The first tool we looked into was Iverilog, as it is open source and provides the basic testing required to check syntax for verilog code. This tool has some very large upsides as it is free and widely used for other similar projects, with proven data. This tool measures syntax correctness by identifying lines of code that don't compile and this incorrectness can be measured via incorrect lines of code divided by total lines of code. The simplicity of this tool made it our initial choice, and with it being open source and free it felt like a clear winner from the start. We also investigated some other syntax correctness tools, with a major one being Verible. This is another open source software that can easily identify syntax correctness and

is widely used. The downside to this software is it was slightly more complicated to set up and was not used due to its increased complexity by similar projects done by companies such as NVIDIA. Concluding our current research on tools to evaluate syntax correctness, we decided that Iverilog was a winner and will be the tool we use going forward unless any major problem presents itself.

Functional correctness is a little harder to test, as different tools have a large gap between how well and in what form they evaluate generated code. This is due to the way that open source tools look for different characteristics. During the first few weeks of our research, the team looked at simulations that could solve Verilog code and evaluate it with impactful feedback to give to the agent framework in order to have better results in future runs. The first idea the team had was Iverilog as using the same evaluation tool for both syntax and functional correctness would make the project much simpler. Throughout our research conducted on Iverilog we believed it would be a simple choice that filled our requirements for the evaluation program. The evaluation done for functional correctness highlights signal issues and outlines where the issues started, which would make it very easy for our agents to evaluate and fix. Iverilog also allows for an output file to be generated that could be read and easily understood by our design agents, making it fill almost every need we require.

During our team meeting with the sponsors on October 12th this old idea of using Iverilog had shown some major complications. Modern design specifications typically use System Verilog to have more advanced designs. This was not a topic we had researched deeply into by the time, and with further research into Iverilog it was discovered that System Verilog is only partially supported. This is being worked on and updated still currently, so by

the end of the design process it could fully support System Verilog, or at least the sections we are looking to incorporate into our design specifications.

System Verilog has some major data types that we would like to include alongside assertions, and specifications such as always_comb and always_ff. These two specifications are specific to System Verilog and allow for hardware design to be more specific and avoid unwanted interactions or undefined X values at the start. Assertions allow us to verify that certain design behaviors contribute to the correct operation. This could be seen in an example such as “Assert(ALU output is valid one clock cycle after input is accepted)”.

This complication of needing to include System Verilog was talked about with the sponsors and although the MVP outlines that our requirement is to only accept and evaluate Verilog code, we believe that accepting System Verilog code is important and should be included. The process of swapping evaluation tools should not be overly complicated, and can be done with a reasonable time frame.

With this new given information, research since has been looking into other evaluation tools and the first that came to mind was Verilator. This tool does a more thorough evaluation of code and has more support than Iverilog. No new dependencies are required when using Verilator and it is also open source allowing for us to not require any funds to evaluate our generated code. While researching this new tool, we figured out it had a similar issue to Iverilog in the way that System Verilog is not fully supported, and although slightly more, it could still run into difficulties. This might not show as we are only using certain parts of System Verilog and will not be fully identified until we start producing code with the framework and agents. One other large

tool we looked into to solve our evaluation issue was Verilog-Eval, yet another free open source tool that evaluates Verilog code. The first thing we looked into with Verilog-Eval is its ability to solve System Verilog and through the documentation it is believed the tool should support all needed requirements. This tool was also designed specifically for evaluating code generated by LLM'S and helping return information that the models can understand and use to generate more correct code.

With all our current given information and research we have decided to begin with Verilog-Eval with a backup plan of using Verilator if required. This helps us attempt to beat our MVP and not just meet the baseline, but also gives a backup plan in the case that we are not able to successfully generate accurate System Verilog code and evaluate it effectively.

The third most important part of evaluation we determined was the amount of human intervention required during each run. This metric, as important as it is, can be skewed not by the LLM's but rather our framework having issues that affect each model differently. The metric of required interventions is still impactful as different models do indeed work better in different frameworks, and that is part of our testing. Each framework tested by various research projects has its own strength and weaknesses, and each large language model thrives in different situations. Unlike syntax and functional correctness, this metric can be tracked by hand, as no tool is required to understand unreasonable amounts of code. This tally of how many times we had to help the system shows how much the agents struggled to either communicate with one another or the models weaknesses in certain design aspects. This evaluation metric will be included in our final report of how each model performs, and any abnormally high number across the board would show a failure on our side for framework design.

7.5 The Evaluation Harness:

The evaluation harness is a standardized way of how to automate our testing for the produced code. This harness is based on the three major metrics we have decided are key to the project. To automate our test for the produced code, the original plan was to feed the code through both Iverilog and Verilog-Eval, in order to get required syntax correctness and functionality correctness. This plan has been altered due to the fact that running our generated code through both programs would require extra time and resources. Verilog-Eval has sufficient capabilities of also testing for syntax correctness, and as of our current research makes the most sense for functionality testing. This would mean our code, after finalized by the framework and final agent, would be fed into Verilog-Eval through a command run by the agent on the server and would result in our final evaluation. One key metric is left out of this automated process in the form of human interventions needed, although as stated earlier this would be tallied by hand during each run and added to a data spreadsheet after each run.

Chapter 8: Project Management and Timeline

Effective project management is critical for a project of this complexity, which blends deep academic research with practical software engineering. Our team adopted a lean Agile methodology to ensure consistent progress, maintain alignment with our sponsors' vision, and adapt to the challenges inherent in a research-heavy development cycle. This chapter details our

team structure, development process, the work completed in the first semester, and our strategic roadmap for Senior Design 2.

8.1 Team Structure and Roles

The project team consists of six team members, each with a designated primary ownership area to ensure clear responsibility and deep expertise in critical components of the system. The Project Manager, Jacobo Forero, is responsible for the overall system architecture and project vision, while the Scrum Master, Dexter Pressley, facilitates our Agile processes in addition to his technical responsibilities. This structure fosters both individual accountability and a highly collaborative environment where architectural decisions are informed by specialized research from all team members.

Name	Official Role	Primary Component Ownership
Jacobo Forero	**Project Manager**	Agentic Core, System Architecture, Schemas, Task Broker
Dexter Pressley	**Scrum Maste**r	LLM Gateway & Provider Abstraction
Mateus Verffel Mayer	Developer	Execution Environment & Tooling (Infrastructure)
Sammy Fares	Developer	Agent Memory & State Management
Andrew Chambers	Developer	Agent Evaluation & Benchmarking
Caleb Elliott	Developer	System Observability & Cost Accounting

8.2 Development Methodology

We employ a lean Agile methodology centered around two-week sprints. This cadence provides a regular rhythm for setting goals, executing tasks, and demonstrating progress. Our Agile ceremonies are consolidated into a single, efficient weekly meeting held on Sundays via Discord, following our session with the project sponsors. This meeting combines elements of several traditional ceremonies:

- Sprint Review & Demo: Team members present the work completed during the previous week to each other and to our sponsors, ensuring continuous feedback and alignment.
- Sprint Retrospective: Following the demos, the team discusses successes, challenges, and potential process improvements to be implemented in the next sprint.
- Sprint Planning: With the previous sprint reviewed, the Project Manager and Scrum Master lead the team in selecting a set of high-priority tasks from the product backlog to form the goal for the upcoming two-week sprint.

For task management, we utilize a streamlined workflow in Jira. Major project initiatives are organized into Epics (e.g., Research, Prototypes, CLI Core Architecture). The product backlog consists of discrete Tasks, each with a clear goal and set of deliverables. This structure has proven effective for our team size, allowing us to maintain clarity and focus without the overhead of more granular subdivisions like stories and sub-tasks.

8.3 Work Completed in Senior Design 1

The first semester was dedicated to establishing a robust foundation for the project, progressing from broad research to the implementation of critical infrastructure prototypes. Our work was structured across four major sprints:

1. Initial Research Sprint: The team conducted a comprehensive literature review of state-of-the-art AI-aided hardware design systems, establishing a shared understanding of existing approaches, their strengths, and their limitations.
2. Specialized Research Sprint: Each team member performed a deep dive into their assigned ownership area, producing the detailed design proposals and research notes that would guide our architectural decisions.
3. Prototyping Sprint: This sprint focused on de-risking the project by implementing isolated prototypes for each major system component. This crucial work validated our technical choices and demonstrated the feasibility of our architecture. Key prototypes developed include:
    - Data Contracts and Task Broker (Jacobo Forero): A full implementation of the Pydantic schemas for TaskMessage and ResultMessage, along with a functional, Dockerized RabbitMQ message bus configured with the required queues, exchanges, and a Dead Letter Queue for fault tolerance. Both components are supported by comprehensive test suites.
    - LLM Gateway (Dexter Pressley): A prototype gateway capable of abstracting calls to multiple LLM providers, with working adapters for OpenAI and a Quinn model.
    - External Tooling Worker (Mateus Verffel Mayer): A prototype worker, packaged in its own Docker image, capable of receiving a task and safely executing an external command-line tool.

- RAG Memory Database (Sammy Fares): A simple but functional prototype of a Retrieval-Augmented Generation (RAG) system, demonstrating the core capability for future agent memory enhancements.
- Observability and Costing Dashboard (Caleb Elliott & Andrew Chambers): A prototype demonstrating the ability to log events from dummy agent runs and visualize them, including cost estimates, in the AgentOps web dashboard.

4. Documentation Sprint: The current sprint, focused on synthesizing our research, design, and implementation work into this comprehensive report.

8.4 Timeline and Milestones for Senior Design 2

Our strategic roadmap for the second semester is designed to build upon the foundational prototypes developed in SD1, systematically implementing the remaining components to achieve our Minimum Viable Product.

Milestone	Timeframe	Goal & Key Deliverables
Foundational Execution & Initial Agents	Weeks 1-4	**Goal:** Build the non-agentic "scaffolding" of the execution phase and begin implementing core agents. **Deliverables:** 1. Implement core Orchestrator logic for DAG traversal and task publication. 2. Implement the Tool Executor and LLM Gateway based on SD1 prototypes. 3. Begin implementation of the Implementation Agent. 4. End-to-end test: Orchestrator publishes a "lint" task, a worker consumes it, runs the linter, and returns a

		result.
Core Generative Loop	Weeks 5-8	**Goal:** Implement the primary code and testbench generation agents to complete the main design flow. **Deliverables:** 1. Complete implementation of the Implementation Agent. 2. Implement the Testbench Agent. 3. Integrate agents with the full orchestration loop. 4. End-to-end test: Provide a simple spec (4-bit counter), and have the system generate and simulate the RTL and testbench.
Advanced Debugging & Analysis Loop	Weeks 9-12	**Goal:** Implement the full Testing_Analysis pipeline for automated debugging. **Deliverables:** 1. Implement the Distillation worker process. 2. Implement the Reflection Agent. 3. Implement the enhanced Debug Agent. 4. End-to-end test: Intentionally introduce a bug and verify the full Distill-Reflect-Debug loop is triggered and attempts a fix.
System Integration, Evaluation & Finalization	Weeks 13-15	**Goal:** Integrate all components, evaluate system performance, and prepare final deliverables. **Deliverables**: 1. Implement the Evaluation Harness and run benchmarks. 2. Integrate the Observability and Cost Accounting pipelines. 3. Final system-wide testing and bug fixing. 4. Write the final SD2 report and prepare the final presentation.

8.5 Conclusion and Future Vision

The work completed in Senior Design 1 has successfully laid the architectural and infrastructural groundwork for our multi-agent system. We have validated our core technical decisions through targeted research and prototyping, and we have a clear, actionable plan for achieving our MVP in the coming semester.

The single most important contribution of our project is its philosophical emphasis on human-in-the-loop, upfront planning. Unlike other systems that focus primarily on the generative capabilities of coding agents, our architecture prioritizes the collaboration between a human designer and a Specification Helper Agent to create an unambiguous plan *before* automation begins. This, combined with our formal artifact state machine, is designed to produce a system that is not only powerful but also transparent, predictable, and trustworthy.

Looking beyond the CLI-based MVP, our long-term vision is to create a product that genuinely augments a hardware engineer's daily workflow. The ultimate goal is an interactive, "Cursor-like" graphical interface where an engineer can converse with the system's agents to collaboratively build and refine the design plan, edit generated code directly, and visualize the entire process. We envision this evolving into a commercial tool for small-to-medium scale FPGA design, with a potential open-source version for academic and research use, ultimately striving to create a tool that engineers trust and find truly effective.